MDRIP: A Hybrid Approach to Parallelisation of Discrete Event Simulation

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science
in the
University of Canterbury
by
Daphne Chao (Yu Fen)

University of Canterbury
2005

Examining Committee:

Prof. Dr. Kryzsztof Pawlikowski, University of Canterbury (supervisor)

Associate Prof. Dr. Don McNickle, University of Canterbury (co-supervisor)

Prof. Dr.-Ing. Helena Szczerbicka, University of Hannover, Germany (external examiner)

Abstract

The research project reported in this thesis considers Multiple Distributed Replications in Parallel (MDRIP), a hybrid approach to parallelisation of quantitative stochastic discrete-event simulation. Parallel Discrete-Event Simulation (PDES) generally covers distributed simulation or simulation with replicated trials. Distributed simulation requires model partitioning and synchronisation among submodels. Simulation with replicated trials can be executed on-line by applying Multiple Replications in Parallel (MRIP). MDRIP has been proposed for overcoming problems related to the large size of simulated models and their complexity, as well as with the problem of controlling the accuracy of the final simulation results.

A survey of PDES investigates several primary issues which are directly related to the parallelisation of DES. A secondary issue related to implementation efficiency is also covered. Statistical analysis as a supporting issue is described. The AKAROA2 package is an implementation of making such supporting issue effortless.

Existing solutions proposed for PDES have exclusively focused on collecting of output data during simulation and conducting analysis of these data when simulation is finished. Such off-line statistical analysis of output data offers no control of statistical errors of the final estimates. On-line control of statistical errors during simulation has been successfully implemented in

AKAROA2, an automated controller of output data analysis during simulation executed in MRIP. However, AKAROA2 cannot be applied directly to distributed simulation.

This thesis reports results of a research project aimed at employing AKAROA2 for launching multiple replications of distributed simulation models and for on-line sequential control of statistical errors associated with a distributed performance measure; i.e. with a performance measure which depends on output data being generated by a number of submodels of distributed simulation. We report changes required in the architecture of AKAROA2 to make MDRIP possible. A new MDRIP-related component of AKAROA2, a distributed simulation engine (*mdrip engine*), is introduced.

Stochastic simulation in its MDRIP version, as implemented in AKAROA2, has been tested in a number of simulation scenarios. We discuss two specific simulation models employed in our tests: (i) a model consisting of independent queues, and (ii) a queueing network consisting of tandem connection of queueing systems. In the first case, we look at the correctness of message orderings from the distributed messages. In the second case, we look at the correctness of output data analysis when the analysed performance measures require data from all submodels of a given (distributed) simulation model. Our tests confirm correctness of our *mdrip engine* design in the cases considered; i.e. in models in which causality errors do not occur. However, we argue that the same design principles should be applicable in the case of distributed simulation models with (potential) causality errors.

Acknowledgements

I would like to thank my supervisors, Professor Krzysztof Pawlikowski and Associate Professor Don McNickle for guidelines on progressing through this thesis work. I also want to thank Dr. Greg Ewing for supporting programming knowledges on the AKAROA2 package. I want to thank Mirko for his time on discussing queueing network systems as well as Lee's time for English spell check. I would like to thank for all my friends who share friendship and time with me. At last, I would like to thank my husband Vincent (Tsu Hen), my Dad, Mum, and brother Yu-Ching for unconditional love and support.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Parallelisation of Discrete Event Simulation

The MRIP (Multiple Replications in Parallel) approach in the AKAROA2 package runs stochastic on-line simulation with non-distributed simulation models [1]. The thesis problem is that MRIP does not support stochastic on-line simulation in distributed simulation which requires a model to be partitioned and distributed. After conducting a survey of parallel discrete event simulation (PDES) and performing some initial testing, the solution to the thesis problem is focused on the MDRIP (Multiple Distributed Replications in Parallel) approach. MDRIP is a hybrid approach to parallelisation of discrete event simulation allowing AKAROA2 to support distributed simulation.

This thesis work has contributed to design and implement MDRIP which consists of *mdrip engine* as well as other associated software, such as *subengine*s. Through this thesis work, it is understood that stochastic on-line simulation for distributed simulation is possible. However, model-dependency remains a problem in design and implementation of *subengine*s as well as how *mdrip*

engine interacts with *subengine*s.

PDES refers to executing Discrete Event Simulation (DES) on multiple processors. DES is a very common scientific methodology across industrial and scientific research areas. As technology evolves and the size and complexity of system models increase, the importance of quantitative output data analysis is increased as well. DES executed on a single processor is no longer sufficient to support such progress. The obvious development is to adopt parallel processing techniques to parallelise DES [4, 5, 17, 30, 36].

The problem of increasing model size comes from the event list. The core operation of a DES lies in an event list that holds unprocessed events. As an event is stochastically created, it is associated with a timestamp and inserted into the event list. The event list processing removes the event with the smallest timestamp value from the event list to run the simulation. Processing the event list is a very time-consuming computational task. As the size of a system model increases, the event list tends to grow proportionally. It is possible that a single processor may run out of computing resource and not be able to finish processing the whole event list [53]. One solution is to decompose a simulation model into smaller submodels running distributedly across multiple processors. Therefore, the main event list is partitioned into several shorter sub-event lists which are more suitable for each single processor to process effectively. Most PDES research [2, 14, 15, 16, 17, 38] is concerned with partitioning a model into submodels, synchronising events among distributed submodels according to sequencing and/or causality constraints, and processing distributed events with timestamps in non-decreasing order.

The problem of increasing model complexity leads to increasing difficulties in understanding system behaviours and analysing system performance. Such model complexity can be reflected by the number of entities in a system

model, the degree of aggregation among entities, and the dynamic nature of a system. Perhaps network protocol design and Internet simulation best describe such a model complexity problem [36, 37]. To ease such problems, distributed simulation allows to observe the interaction between submodels and also enables performance evaluation at various granularity.

It has been pointed out that lack of quantitative and sequential analysis on simulation output data results in a credibility crisis [3, 7]. Although statistical inference is part of very elementary academic scientific training, still many studies are conducted without proper statistical analysis. Output data analysis is usually conducted in an off-line fashion in that simulation output data is processed after the simulation is finished. Thus, off-line simulation comes with a fixed run length. The problem is that off-line simulation with a fixed run length provides no chance of meeting pre-set confidence intervals [29]. Therefore, proper statistical inference requires that analysis is run on-line so that adjustable accuracy levels and confidence intervals are adaptively and sequentially controlled. The AKAROA2 package is an implementation which can automate output data analysis with either fixed or flexible run lengths [3, 4, 5, 6, 7, 8].

On-line simulation may expect very long simulation run lengths in order to produce sufficient numbers of observations for a required accuracy level. In AKAROA2, MRIP is designed and implemented to run the simulation of the same model with different streams of random numbers on multiple processors. As the simulation run length from each single processor is added up by multiple replications, an overall simulation speedup is achieved. Therefore, run times are improved. Several experimental studies using MRIP in AKAROA2 show promising results [3, 4, 5, 6, 7, 8, 22, 23]. Other similar MRIP studies can be found in [10, 11, 55].

The MRIP approach does not require model partitioning. Each simulation run simulates a system model as a whole without model partitioning and distribution. MRIP deals with multiple replications of non-distributed simulation models. On the other hand, distributed simulation deals with a single replication of a distributed simulation run. MRIP easily achieves more efficient statistical speedup than distributed simulation, because simulation run length of distributed simulation is not multiplied. Moreover, model-dependent synchronisation overheads may be very expensive for the distributed simulation itself [5, 16]. Distributed simulation explores parallelism of simulation models and focuses on partitioning a model into submodels and synchronising event messages among submodels. MRIP is fault tolerant and consumes more memory resources than distributed simulation, whereas distributed simulation is more useful to the problems related to model size and complexity.

AKAROA2 as a PDES implementation of the MRIP approach is currently limited to non-distributed simulation models only [1]. To improve the functionality of AKAROA2 with regard to increases of model size and complexity, this thesis designs and implements a Multiple Distributed Replications in Parallel (MDRIP) approach. This new MDRIP approach allows us to run distributed simulation under the quantitative and sequential control of MRIP.

The following section describes requirements of this new MDRIP approach.

1.2 The MDRIP Approach

For this thesis, a new PDES approach is proposed and implemented: Multiple Distributed Replications in Parallel (MDRIP).

The objective of MDRIP is to establish the provision of quantitative and sequential control of distributed simulation under MRIP. The concept of MDRIP is to combine Multiple Replications in Parallel (MRIP) and distributed simulation.

Previous studies have mentioned similar ideas [5, 6, 12, 13]. Ewing, McNickle, and Pawlikowski discussed the possible theoretical speedup of combining both MRIP and distributed simulation in [5, 6]. Heidelberger mentioned an alternative in theory to combine replications and distributed simulation to reduce bias as well as improve efficiency [12, 13]. However, this thesis work is the first attempt to design and implement the MDRIP approach.

The following aspects identify what a MDRIP approach consists of in the context of AKAROA2.

- **Automation of Output Data Analysis**

  Automation of output data analysis in AKAROA2 refers to quantitative and sequential control of accuracy levels of statistical errors and confidence intervals of output results. Such automation reinforces the capability for statistical inference.

  From MRIP to MDRIP, the automation of output data analysis is extended from simulation *engine* to *mdrip engine*, which manages distributed simulation running on *subengines*. As different streams of random numbers are dispatched to distributed *subengines*, distributed data observations are produced when the *subengines* receive the dis-

tributedly allocated random numbers. Each *subengine* sends its data observations back to its associated *mdrip engine* that collects and coordinates distributed data observations.

As a result, MDRIP also reinforces statistical inference by automating output data analysis.

- **Logical Processes (LPs)**

  Most PDES research uses the term - Logical Process (LP) - to describe a disjoint set of the whole simulation model running on one single processor [16, 17, 19, 20]. Such a disjoint set, often equivalent to a submodel or a set of some submodels, is itself a sequential computation unit with local event list and local clock time. In this thesis, one LP represents one distributed simulation *subengine* simulating a submodel executed on one particular processor.

  The development of MDRIP follows such term and definition - Logical Process (LP) - to be able to transform model-related issues transparently from distributed simulation to MDRIP.

- **Random Numbers**

  The MRIP approach under AKAROA2 simulates a model with multiple runs using different streams of random numbers. Each simulation run is handled by a simulation *engine*. Independent runs refer to using different streams of random numbers for each simulation *engine*. Identical runs refer to the same simulation model for each simulation *engine*. Thus, after multiple replications, output data from different *engines* is guaranteed to be uncorrelated [5, 6, 7].

  Figure 1.1 shows how random numbers work under MRIP in AKAROA2. R_{1i} represents the i-th random number in a sequence of random num-

ber stream for the first simulation run. R_{ni} represents the i-th random number in a sequence of random number stream for the n-th simulation run.

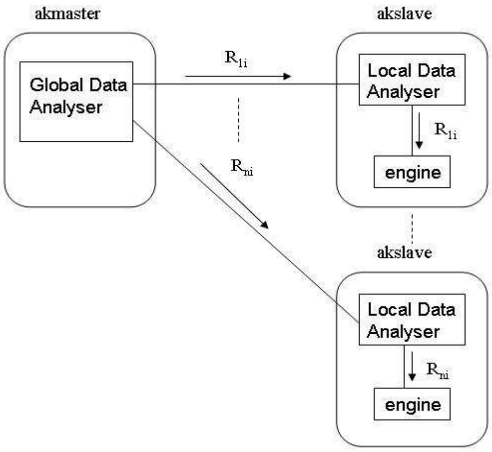

Figure 1.1: Random Numbers in MRIP

To extend MRIP to MDRIP, the MRIP approach mentioned above is maintained with each run developed further into a distributed simulation. With each simulation run as a distributed simulation, the target simulation model is designed to be partitioned into submodels handled by *subengines* running distributedly on multiple processors. Each *subengine* of a simulation run requests a stream of random numbers via the *mdrip engine* of that simulation run from random number generators managed by *akmaster* in AKAROA2. Upon receiving such a request, *akmaster* then allocates a block of random numbers via

the *mdrip engine* and dispatches such random numbers to associated *subengines*. As a result, each *subengine* uses different streams of random numbers to simulate its local events. Thus, output data from different *subengines* is also guaranteed to be uncorrelated.

Figure 1.2 shows how random numbers work in the extension to the MDRIP approach. Similarly, R_{1xi}, R_{1yi}, and R_{1zi} represent the i-th random numbers which are part of the sequences of random number stream for each *subengine* x, y, and z of the first simulation run. R_{nxi}, R_{nyi}, and R_{nzi} represent the i-th random numbers which are part of the sequences of random number stream for each *subengine* x, y, and z of the n-th simulation run.

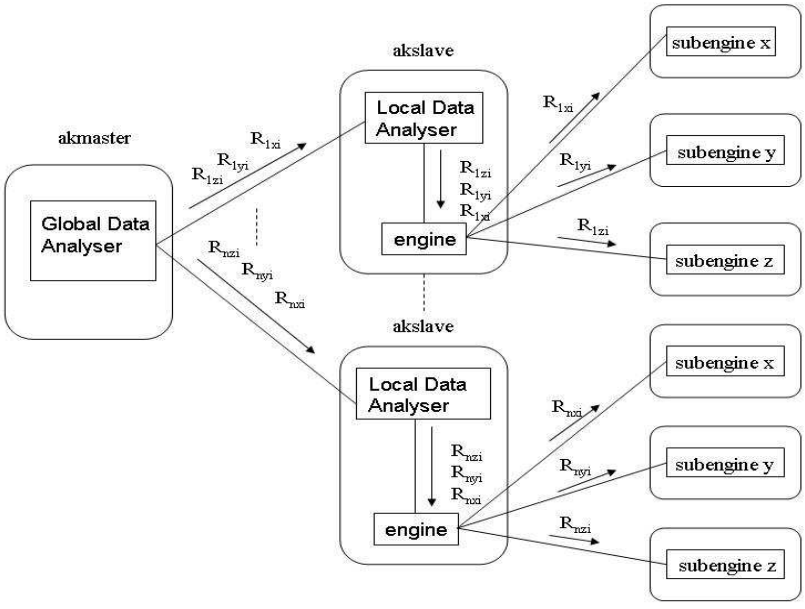

Figure 1.2: Random Numbers in MDRIP

- **Newly Distributed Messages**

  The MDRIP approach in AKAROA2 requires two types of distributed messages: random numbers and data observations. The messages of random numbers maintain the stochastic characteristics of the MDRIP approach. The messages of data observations are submitted to *akmaster* via the *mdrip engine*. Such data observation messages supply raw data for on-line output data analysis.

- **Parallelisation Platform**

  MDRIP is a hybrid approach to parallelisation of discrete event simulation. Features related to parallelisation platform need to be examined. As far as the implementation is concerned, MDRIP is extended from MRIP. Therefore, parallelisation platform of MRIP is followed to ensure overall compatibility. Relevant features include hardware configuration, operating system, AKAROA2 version, networking protocol, and interprocess communication.

  General information about hardware configuration, operating system, AKAROA2 version, and networking protocols is listed as follows:

  - AKAROA2 package version: 2.7.5.
  - Operating System: Linux Red Hat 3.4.2-6.fc3
  - CPU, Cache, and Memory:

	CPU	Cache	Memory
cosc4xx	Intel(R) Pentium(R) 4 CPU 2.80GHz	512 KB	1024 MB
cosc3xx	Intel(R) Pentium(R) 4 CPU 2.40GHz	512 KB	512 MB
cosc2xx	AMD Athlon(tm) XP 1600+	256 KB	512 MB

  - Network: TCP/IP over Ethernet

– Interprocess communication: Message Passing via I/O Multiplexing

In summary, this thesis work contributes to the design and implementation of the MDRIP approach in AKAROA2 which extends the current version of simulation *engine* into *mdrip engine* with the capability of launching simulation *subengines* and distributing submodels on multiple processors, as well as retaining central management of quantitative and sequential control.

1.3 Thesis Layout

Chapter 1 gives an introduction on parallelisation of discrete event simulation in general and discusses the requirements of the new hybrid MDRIP approach. Chapter 2 reports a survey on previous PDES activities and discusses important PDES issues. Chapter 3 outlines design of the MDRIP approach with overviews on various parallelisation approaches, networking architecture, and system components. Chapter 4 describes implementation of the MDRIP approach in details. Chapter 5 discusses testing efforts for the development of the MDRIP approach. Chapter 6 provides conclusions and suggests future work.

Chapter 2

Survey of Parallel Discrete Event Simulation

This survey investigates important issues related to PDES. Dating back to early 80s, PDES emerged as a research field on the execution of discrete event simulations on parallel/distributed computers [15, 16, 17].

For this thesis, eight issues have been identified with six are the primary issues, one is the secondary issue, and the other one is a supporting issue. The primary issues are directly related to the problems that can possibly arise if the parallelisation of DES is to take place. For examples, model partitioning, load balancing, interprocess communication, synchronisation, interoperability, and performance studies. Among them, model partitioning, load balancing, and synchronisation issues are model-dependent which means solutions can be different from model to model. While issues, such as interprocess communication, interoperability, and performance studies, are less model-dependent so that considerations may generally be shared by all models.

Event list management is the secondary issue that is due to the algorithm

efficiency of the event list implementation, rather than directly related to the parallelisation of a DES.

One supporting issue addressed is statistical analysis which is either processed on-line or off-line. Off-line simulation features fixed run length. However, off-line simulation provides no sequential control. On-line simulation is particularly important to support better statistical inference on simulation research. The AKAROA2 package [1] dedicates in stochastic on-line simulation and provides automated solutions on output data analysis to such supporting issue.

After discussion on relevant issues of PDES, some thoughts on the MDRIP approach are mentioned. PDES does not necessarily assume that statistical analysis is adaptively controllable. However, as far as MRIP and MDRIP are concerned, the central design concept is to employ adjustable statistical accuracy with distributed simulation.

2.1 Issues

2.1.1 Model Partitioning

A simulation model is partitioned into multiple loosely coupled submodels according to certain world view relationships. Each submodel is represented by an LP or a set of LPs running on one processor. The simulation is run by executing several submodels in parallel. These submodels are distributed and simulated on multiple processors by communicating with each other via interprocess communication mechanisms, such as message passing or shared variables.

The major goal of model partitioning is to exploit the inherent parallelism in simulation models in order to take advantage of parallel processing to

accelerate the simulation execution time [17, 65]. It requires certain domain knowledge to be able to partition a simulation model effectively. Moreover, it requires a certain understanding of PDES to be able to identify inherently partitionable factors for the purpose of efficient PDES.

1. **Graph partitioning**

   Graph theory is often used to assist analysis in the modelling phase, especially in VLSI (Very Large Scale Integration) simulation [65, 71]. In general, graph partitioning refers to mapping a system model into a partition scheme that groups strongly connected nodes into logical blocks so that each block is about the same size with minimum communication links. Each block is mapped to one processor and the communication links between the blocks represent communication overheads. A simulation model represents a Problem Graph (PG). A set of nodes represents a set of LPs, while a set of links represents a set of communication channels. The node weight features the expected execution time for the associated LP. The link weight features the amount of message traffic expected over such link. The link direction features the logical precedence of the nodes in a PG [42].

   Nandy and Loucks [65] discuss partitioning quality, mentioning how a model partitioning scheme affects the overall PDES performance. If the conditions of interprocess communication and load balancing are the same, different graph partitioning schemes have different performance results. A good model partitioning scheme is expected to result in low overhead of interprocess communication and well-balanced process workloads among distributed processors. Otherwise, the performance of parallelisation may be affected.

2. **Static vs Dynamic partitionings**

   Whether a partitioning scheme is allowed to be changed or not during a PDES run distinguishes static partitioning from dynamic partitioning. Static partitioning is sometimes called sequential partitioning in that a partition scheme is available prior to the PDES execution and remains unchanged throughout the entire PDES execution. While dynamic partitioning prepares an initial partition scheme and keeps modifying the scheme in progress throughout the PDES execution. Static partitioning is inflexible towards workloads balancing. However, dynamic partitioning may introduce migration overheads caused by adjustable workload balance [42].

Model partitioning is not a straightforward task. It requires extensively iterative or refined processes. The efficiency of model partitioning is concerned with trade-offs between reduction of intercommunication costs and adequately distributed workload balance. The outcome of the model partitioning determines the structures of communication topology and load balance which instead affect overall PDES performance.

2.1.2 Load Balancing

Load balancing refers to the distribution of the workload among multiple processors where the workload means the number of events to be processed [35]. The aim of load balancing is to arrange evenly distributed workloads, or at least as evenly as possible, so that each processor is more efficiently utilised and less idle. The idea is that higher processor utilisation results in more efficient PDES.

Load imbalance may significantly degrade PDES performance due to idle processors. However, the trade-off between load balancing and communica-

tion costs has never been a simple task. *Perfectly-balanced* workload may imply high communication costs. In order to minimise such communication costs, PDES designs may need to compromise on load balancing and allow certain idle times [35].

1. **Static load balancing**

   If sufficient information of estimated workload is available before the PDES simulation execution takes place, then static load balancing can be applied. Nandy and Loucks [65, 68] apply static load balancing together with conservative synchronisation using null messages.

   If the event workload is varied during runtime, static load balancing will not have the flexibility to adjust to such variation. As a consequence, some processors may be heavily utilised, while some other processors may be idle. Lack of adjustability is a weakness especially in simulating dynamic systems. Such weakness may consequently degrades PDES performance. Furthermore, if information of estimated workload is theoretically or experimentally inaccurate, then a PDES is actually unbalanced right from the beginning [65].

   Based on Time Warp, Nicol and Reynolds [66, 68] work on development of load balancing from static to dynamic.

2. **Dynamic load balancing**

   If sufficient information on estimated workload is not available before the execution of PDES takes place, or if workloads may vary, then dynamic load balancing is suitable in that workload distribution is decided during runtime.

   LPs distributed among multiple processors are migrated from one processor to another according to variations of event workload. Such a

dynamic feature is suitable for dynamic simulation models. For example, Gan, et al., [35] uses dynamic load balancing in a supply chain simulation model to keep inventory control at a balanced level.

When discussing optimistic synchronisation, Reiher and Jefferson [67, 68] emphasise on effective processor utilisation, instead of high processor utilisation, in that only actually committed computations are taken into account. High processor utilisation does not necessary mean good performance, if busy workload will possibly be rolled back later due to optimistic synchronisation. Because process migration may generate high communication overheads from historical state-saved data, Reiher and Jefferson suggest phase splitting [67, 68] to distinguish a process between the old phase for historical state-saved data and the new phase for message passing action itself. Only data in the new phase is processed at the destined processor. Therefore, performance is improved.

Instead of process migration, Schlagenhaft et al., [71] developes cluster migration to adaptively control when, what, and where workload processes are migrated over different processors. Improved performance is reported with reduced rollbacks.

2.1.3 Interprocess Communication

Various aspects of interprocess communication related to PDES cover communication topology, data exchanges (shared variables vs message passings), underlying hardware platforms, geographical coverage, latency, bandwidth, as well as networking types. Decisions made related to these aspects of communication infrastructure are important in building effective PDES [16, 17].

1. **Communication topology**

   The outcome of a model partitioning scheme forms the communication topology of a PDES. Such topology lays out the overhead structure of interprocess communication. In the case of static partitioning, the communication topology is fixed. Otherwise, the communication topology will keep changing during the PDES execution. Decisions on load balancing and synchronisation protocols further improve or aggravate such overheads. The overhead of interprocess communication is inevitable and should be minimised.

2. **Shared variables vs message passings**

   There are two main approaches: shared variables mean that defined variables are accessible by different processors. On the other hand, message passings do not allow variables to be accessed by the other processors, so that intercommunication among multiple processors is by sending or receiving messages explicitly.

3. **Underlying hardware platforms**

   The real time bounds of interprocess communication are determined by the underlying hardware platforms. A brief taxonomy in Figure 2.1 on classification of parallel and distributed computers is taken from [17].

   - **Shared-memory machines**

     Multiple processors access memory through a connected high-speed switch. High-speed cache memory attached to each processor stores frequently used data and instruction. Either shared variables or message passings can be used.

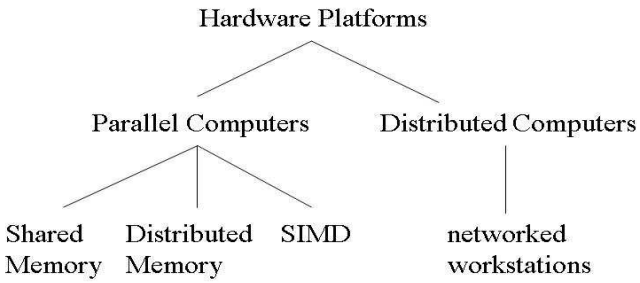

Figure 2.1: A Brief Taxonomy of Parallel and Distributed Computers

- **Distributed-memory multiprocessors**

  Each processor unit consists of a CPU, cache memory, main memory, and communication controller handling explicit message sends and receives. Only message passing is possible. Here, cache memory holds only local data and main memory is only local to each processor.

- **SIMD**

  Referring to single-instruction-stream, multiple-data-stream, a parallel processor uses a single instruction stream to control multiple data streams.

- **Distributed computers**

  The most popular hardware platform is characterised as interconnected networks of stand-alone computers. Commonly used operating systems include Linux, Unix, and Windows. Distributed

computers are heterogeneous in two ways: one is of different manufacturers, the other is of different switching facilities, such as asynchronous transfer mode (ATM) or Ethernet.

It is noted that differences between parallel and distributed computers are blurred. It is meaningless to draw a sharp line between parallel and distributed computers. Other features characterise underlying hardware platforms are more important [17].

1. **Geographical coverage**

   How the range of the PDES execution of distributed submodels is physically located. Such a coverage can be networked machines in a single machine room, or distributed over the Internet across different countries. The details of geographical coverage set out the overhead of latency.

2. **Latency**

   Latency defines the communication delay measured by the time needed to transfer a message from one processor to another. The common unit of time is microseconds. Parallel computers normally have less than 100 microseconds latency, while distributed computers have hundreds of microseconds. Satellite link is an example of very long latency which may take up to seconds. On the other hand, Fujimoto et al., [72] designs a rollback chip to empower hardware magnitude of very short latency over extremely heavy used communication links such as for state saving and rollback [68].

3. **Bandwidth**

   Bandwidth refers to the capacity of a communication channel which is hardware dependent. In general, the more bandwidth available, the

less significant interprocess communication overheads become.

4. **Networking types**

   In heterogeneous networks, the measures of bandwidth and latency are varied. While in homogeneous networks, such measures are near constant factors.

2.1.4 Synchronisation

As DES is developed into PDES and distributed events are processed on multiple processors, causality constraints may arise because the future can not be affected by the past [16]. Synchronisation or time management to maintain such causality constraints contributes to the most dominant communication cost of PDES. Consequently, the majority of PDES research has focused on development of different synchronisation protocols with an aim to improve simulation efficiency [14, 15, 16].

Conservative approaches strictly forbid causality errors. Optimistic approaches conditionally allow causality errors to happen, but will recover such errors and re-run simulation since then. Hybrid approaches attempt to combine advantageous aspects of both conservative and optimistic approaches [15, 16].

1. **Conservative Approaches**

   Conservative approaches need to decide when it is safe to process an event. Lookahead is one particular crucial value which defines when the next future event is likely to happen [16, 17]. If an LP contains an unprocessed event in its local event queue, according to the timestamp of this unprocessed event, this LP needs to decide that it will not later receive another event with smaller timestamp value. Then this unprocessed event is safe to be processed. If the LP can not decide

whether this unprocessed event is safe or not, the LP needs to be blocked, which may result in deadlock. How to avoid or detect such deadlock is the main concern for conservative approaches.

The earliest PDES synchronisation protocols, taking conservative approaches, were independently developed by Bryant in 1979 [39] and by Chandy and Mirsa in 1986 [40]. Static communication topology was assumed.

- **From null messages to deadlock avoidance**

  Initially, Chandy and Mirsa used null messages to avoid deadlocks. Sending a null message after each finishing process may generate many unnecessary overheads. The CMB protocol refers to the first and very basic conservative synchronisation approach of PDES [39, 40].

  A null message is a do-nothing message. After an LP finishes processing an event, it sends a null message to notify all connected nodes with the finishing time as the timestamp value of such a null message. Null messages can avoid deadlock. The biggest drawback comes from the communication overhead of the null message itself, because the null messages associated with correct events can be a majority which is simply unnecessary and uneconomical.

  Later, Chandy and Mirsa shifted to detect deadlocks rather than to avoid them. Once deadlocks are detected, they are broken and recovered. However, overheads are still a problem.

- **SRADS**

  Reynolds [73] proposes SRADS protocol in which null messages sent as an on-demand basis can reduce redundant communication costs. When a receiving link with the smallest timestamp runs

out of messages to process, which indicates a process is about to block, a request next message is sent only to the link at the sending side of the link. Although the time taken to receive a null message by request is doubled due to request transmission and send transmission [43].

- **Deadlock detection and recovery**

  Deadlocks are not avoided. However, detection steps are taken to discover when a simulation is deadlocked. Once a deadlock is detected, it needs to be removed. A problem is that the size of distributed networks can be too large to efficiently detect deadlocks in subnetworks. So, preprocessing deadlocks in each subnetwork can be useful [44].

- **Conservative time windows**

  Significant searching is required to determine if an event is safe to process. To perform less searching, a range of LPs is set to search for the next unprocessed event from the range of LPs. It is model dependent to decide the bounds of the time windows. If the range is small, the PDES is less parallel. If the range is too large, it does little help to reduce searching costs. The size of such a time window is referred to as the 'bounded leg' by which the minimum distance between LPs is used to determine if it is safe to process an event [45].

- **Conditional knowledge**

  Events are differentiated into definite events and conditional events. It is always safe to process definite or unconditional events, while related predicate needs to be satisfied first in order to convert a conditional event to a definite event. Such predicate is arranged

into processes as part of passing messages. If a conditional event contains the smallest timestamp, it is still safe to process [46].

2. **Optimistic Approaches**

   Optimistic approaches need to detect when a causality error has happened and recover such an error by rolling back previously processed events. Rollback involves either regular state saving or sending a negative message (anti-message) to replace the original message. Normal event messages refer to positive messages [16].

   Jefferson [41] borrowed ideas of paging or segmentation from virtual memory to define Virtual Time as a global, one-dimensional, temporal system coordinates timestamps of distributed events. Virtual Time was implemented in Time Warp which is the first PDES that synchronises distributed events optimistically.

   Global Virtual Time (GVT) refers to the smallest timestamp among all unprocessed messages. GVT is a lower bound for determining rollback actions. To compute GVT periodically is memory intensive, because such periodical computation requires huge state saving data. State saving is itself a programming practice consuming lots of memory.

   - **Time Warp**

     If an event message with timestamp smaller than the timestamp of the last processed message is received, then a causality error is detected. Such a received message resulting in rollback is a straggler message. All previous events related to the received message causing a straggler need to be undone [41].

   - **Georgia Tech Time Warp - GTW**

     Led by Fujimoto, Georgia Tech Time Warp (GTW) was developed

to include various optimistic synchronisation techniques, such as direct cancellation, advanced GVT computation, fossil collection on-the-fly, etc., to optimise the cancellation of incorrect messages. A graphical visualisation system for general purpose network computing simulation, PVaniM, adds animation features for more insight into advanced GTW [2, 17]. A hardware solution with a *rollback chip* is designed to improve demanding memory requirements from state saving and rollback [68].

- **Lazy cancellation**

  Gafni states that fix an incorrect timestamped message is cheaper than to totally discard it. Only when the re-executed processes make sure incorrect answers are produced, thus anti-messages are sent to request rollback. Due to the recursive nature of rollback, lazy cancellation is very sensitive to model dependency [47].

- **Lazy re-evaluation**

  West [48] compares the state vector of a process before and after a straggler event which is an event violating causality constraint [17]. If the state has not been changed, it means such a process is a correct event. Then there is no need to roll back, but to skip the roll back and jump forward. Lazy re-evaluation works particularly well with read-only or query events. However, it can be difficult to implement and maintain.

- **Moving Time windows**

  Sokol et al., [49] examines events with timestamps within a specific time frame to see if incorrectness has been propagated. If no incorrect event is found within a specific time frame, then such examinations just degrade the overall performance. In addition,

there is no logical way to determine the period of such a time frame.

- **Direct cancellation**

  Instead of searching for where to cancel incorrect messages, anti-messages are given higher priority than positive messages [51]. Fujimoto states the "dog chasing its own tail" effect in Time Warp can be avoided as well as the costs to cancel incorrect messages are reduced [17, 51].

- **Wolf calls**

  Madisetti, Walrand, and Messerschmitt [50] use the straggler message to embed control logic to notify messages infected by incorrect events. However, correct events may be idled, too. Thus performance is further degraded. It is difficult to implement such embedded control logic, because certain real-time values are not feasible to obtain.

- **Space-time graph**

  Chandy and Sherman [52] utilise a two-dimensional space-time graph consisting of state variables on one axis and simulation time on the other axis. Disjoint regions are partitioned and represented by LPs. LPs fill in their assigned regions and exchange messages by updating boundary conditions. Until a fixed point is computed, the message exchange is stopped. New messages sent to other LPs will update boundary conditions. Such two-dimensional space-time style shares the same concept of relaxation from continuous simulation. In PDES, however, the time-space area is specially defined as a rectangle for each LP [17, 52].

- **Memory management**

  Memory usage is a critical concern in particular for optimistic synchronisation, because GVT and possible rollback computations need huge memory resources. If all LPs have enough memory to use, then that is not an issue. Once an LP runs out of memory in its local processor, then the memory stall occurs resulting simulation stall and incomplete. The objective of memory management is to delay the possible occurrence of memory stall [57].

  The solution to memory management depends on whether to guarantee sufficient memory usage or to improve simulation completion ratio by adapted heuristics. The guaranteed approach assumes the same amount of memory in both parallel and sequential executions and needs to find out the minimum required memory for a specified synchronisation protocol. In order to provide such guarantee, sufficient memory is allocated and fixed. As a result, execution time may suffer. However, any other adapted approaches may run faster, but do not guarantee to complete the simulation [57].

  Fossil collection is performed to reuse memory and process undoable I/O actions. Batch fossil collection reclaims memory periodically by searching through the sub-event lists of related LPs. Such searching can be time-consuming. An extra FIFO queue is arranged to hold processed events. On-the-fly fossil collection reclaims memory from this FIFO queue only if required [16, 17].

  If the size of the state and the number of states that must be saved can be reduced, then less required memory will result in a better situation to prevent unwanted memory stall.

Memory will eventually stall. Jefferson [57] applies **cancelback** to send back messages to LPs for memory recovery. Preiss and Loucks [56] use **pruneback** to delete selected previously saved states for recovering memory space.

- **Rollback relaxation**

  When rollback occurs, only local recovery is performed. I/O events are not synchronised. If basic definition of causality is insignificant, then causality is not strictly obeyed. Therefore, in a tactical sense, the final result will not be affected. This is a starting point to tradeoff between causality maintenance and overall PDES performance. It leads to unsynchronised approaches [31].

3. **Hybrid Approaches**

   Hybrid-styled PDES solutions are considered to be the future direction [17]. The ideas to mix conservative and optimistic approaches take enhanced performance into consideration.

   - **Filtered rollback**

     Lubachevsky, et al., [53] combines conservative bounded leg and the optimistic moving time window, while the causality constraints are violable and the minimum distances between LPs are adjusted in order to optimise PDES.

   - **Switching SRADS**

     Also known as SRADS with local rollback, Dickens and Reynolds [75] process an event conservatively in an LP until there is no safe event, then switch to process an event optimistically, but possible rollbacks are limited locally only in that LP. Such local rollbacks approach is described as aggressive [14, 54].

2.1.5 Event List Management

Various studies on data structures based on priority queues have been dedicated to improving the performance of DES [58, 59, 60, 61]. The basic event list management involves inserting an event into a queue (enqueue), sorted and ordered by timestamps, and removing the event with the smallest timestamp from the queue (dequeue or delete-minimum). Hence, the per event cost consists of inserting and removing an event from the event list. If the focus is shifted to PDES, not only such two basic operations (enqueue and dequeue) are important, but also fossil collection and rollback require to keep track of processed and unprocessed events. As a result, data structures are modified to allow for such PDES conditions.

An event list implemented with efficient data structures can keep the secondary effect to a possible minimum in order to prevent runaway processes. Such secondary effect refers to the use of inefficient data structure for simulation implementation that causes much worse performance degradation than causing simply by increased search time.

Several synthetic benchmarks for performance evaluation are developed. Based on synthetic workloads, these benchmarks tell how efficient a pending event list is implemented [59, 62, 63, 64]. If the implementation of a pending event list has not been tested under any performance benchmarks, then the final performance evaluation of PDES will be unreliable.

1. **Pending Event List**

   The core of discrete event simulation (DES) lies in managing a pending event list (or set) that stores future events with simulation timestamps. Events model the system changes occurred at discrete points in time. The change of a timestamp means the change of the system state. This is usually implemented as a priority queue with timestamps as the keys

and simulation estimates as the values. Event list management is actually a sorting computation that is a very time-consuming computing task and is the dominant computing cost of a DES. Efficient event list processing is crucial to DES performance. PDES presents more challenges. For example, Time Warp needs to consider both past and future events because rollback is likely to happen.

2. **Data Structures**

   Assumed the number of event is N, it has been shown that a sorted linked list costs $O(N)$ to insert an event and costs $O(1)$ to remove an event. A heap costs $O(logN)$ to both insert and remove an event. Calendar queues costs $O(1)$ on average to insert and to remove an event. Johns [58] experiments event list with two-list and Henriksen's event-set implementations. Event list management of DES provides special cases for priority queue research.

   Taken PDES into account, Ronngren et al., [59] has improved skew heap of $O(log(N))$ execution time. Skew heap is an ordered binary tree that descendant has lower priority. Meld operation merges two skew queues into one and the heap property is preserved. Also lazy queue [59] is a multi-list with average queue access time of $N(1)$ and worst case of $O(log(N))$. Brown [61] utilises another multi-list - calendar queue with average queue access time at $N(1)$ and worst case of $O(N)$.

3. **Synthetic Benchmarks**

   Synthetic workloads with varied message populations, queue sizes, and numbers of unprocessed messages, are designed to test the efficiency of data structures implemented for pending event list. In particular, secondary effect should be isolated and not confused with overall PDES performance analysis.

- **From Conventional Hold to Generalised Hold**

  Vaucher [64] generates the conventional Hold model with a synthetic workload of long sequences of events performed on a fixed size queue. Thus the average time per event operation is a function of the queue size. However, such fix scheme does not reflect the performance degradation problems. Chou et al., [63] presents a more realistic generalised Hold model in that the size of the pending event list is not fixed.

- **PHOLD**

  Fujimoto [62] proposes homogeneous workload with even message density for each LP. The size of the pending event list is predictable.

- **Arbitrary Flow Network Model**

  Heterogeneous workload is designed with varied message density for each LP and the size of the pending event list can be increased significantly. Such synthetic workload serves as a stress test, if the efficiency of data structure is of major concern [59].

2.1.6 Statistical Analysis

Statistical analysis on simulation output data is generally considered as a supporting issue. Off-line output data analysis is a major method among simulation research. Such off-line simulation decides in advance how long the simulation should run, therefore the simulation run length is pre-set and fixed. On the other hand, on-line data analysis during simulation allows simulation run length to be decided during the simulation. During such sequential simulation, confidence intervals are tested to see whether accuracy criteria of simulation are satisfied.

1. **Statistical Variability**

   Sauer and MacNair [28] consider that DES is coupled with statistical variability from the use of random number streams. Pawlikowski [8] states "Statistical inference is an absolute necessity in any situation when the same (correct) program produces different (but correct) output data from each run. Any sequence x_1, x_2, ..., x_n of such output data simply consists of realisations of random variables X_1, X_2, ..., X_n".

2. **Confidence Intervals (CIs)**

   In analysing DES or PDES simulation output data, errors due to the statistical variability should be analysed. CI is an estimated range of statistical value for an unknown parameter, for example the mean value. If the CI is 95% and the width of CI takes 5%, then the true estimate mean value may most likely be contained in the range of (92.5%,97.5%).

3. **Sequential Control**

   The AKAROA2 package [1] automates the sequential control by adjusting pre-set confidence level allowing more exhaustive examinations in simulation analysis [6, 7, 8]. Sequential control applies the stopping rules to adaptively adjust levels of statistical accuracy [28]. The stopping rule implemented in AKAROA2 is formulated as follows:

   - δN - the current relative error of results

     δ_{max} - the maximum acceptable relative statistical error

- Given δ_{max}

  if $\delta N > \delta_{max}$, then simulation is continued until the next checkpoint

  if $\delta N \leq \delta_{max}$, then simulation is stopped

If the specified accuracy is reached, the simulation is terminated. If the specified accuracy is not reached, the simulation is continued until the specified accuracy is reached or the processors run out of processing resources [1, 6, 28].

2.1.7 Interoperability (Extensibility)

Interoperability or extensibility of PDES covers state-of-the-art developments like federated simulations, reusability, High Level Architecture (HLA), and web-based simulation [17]. The hybrid MDRIP approach has the potential to interoperate with these developments. Interoperability focuses on seamless PDES among different simulators. Extensibility looks at how a simulator extends its work to co-simulate with other simulators. Interoperability and extensibility share similar issues in PDES, thus, this section puts them together for discussion. Network researchers, Bajaj et al., [30] identifies extensibility as one of the five simulation needs. Component-based PDES design with composable modeling is essential to achieve effective interoperability.

1. **Federated simulations**

   Federated simulations refer to different simulators as different federates working together under a global conceptual model of an entire simulation as a federation. Run time infrastructure (RTI) is designed for time management required by such federated interoperation [17].

Proxy concept is adopted to link complicated mappings between federates [19].

To be consistent with PDES, one federate is treated as an LP. Synchronisation follows either conservative or optimistic approaches. Simulation federation entails PDES in heterogeneous networks [34].

2. **Reusability**

   In terms of both modeling and software engineering practices, reusability is highly desirable for PDES. Good reusability not only saves development budget and time, but also encourages more interoperable or extensible PDES activities. However, industry sectors tend to be reluctant in reusability due to marketing and administrative reasons.

   To design and validate a correct model is not a simple task. In the case of large and complex systems, it is not feasible to develop everything from scratch. For example the NS2 in network protocol design, the split-programming model adopts C++ implemented in simulation kernel and OTcl scripting language implemented in developing simulation models as well as configuration and control of simulation run. In addition to freely available libraries of protocol scripting, the object oriented features of both C++ and Otcl further enhance reusability of various networking models. And such reusability further advances the overall network simulation research [30].

   Reusability of PDES functions emphasises modular and component-based design with well-defined interfaces. For example, through the AKAROA2/NS2 interface, one can perform sequential controlled MRIP simulation with many NS2 network protocols [22, 23]. RTI is another example where the time management functions of PDES are reused [2, 17, 18, 19, 20, 69].

3. **High Level Architecture (HLA)**

   HLA is a set of rules specifying for federated simulations. These rules cover IFSpec (Interface Specification) and OMT (Object Model Template). OMT includes FOM (Federation Object Model) and SOM (Simulation Object Model). RTI (Runtime Infrastructure) defines a software environment for distributed federates. One simulator needs to implement its RTI functions to be able to federate with the other simulator interoperating through RTI [17].

   Dating back to 1983, the SIMNET (SIMulator NETworking) project was designed for military training in virtual environments. After SIMNET, Distributed Interactive Simulation (DIS) defined standards of interoperability regarding geographically distributed and autonomous simulators. The Aggregate Level Simulation Protocol (ALSP) stemming from DIS brought PDES to war game applications [17].

   Began in 1995, High Level Architecture (HLA) defined PDES baseline that DoD requires HLA compliant simulations. Previous PDES efforts from DIS and ALSP were merged into an integrated and interoperated simulation environment. The synchronisation in such PDES baseline defined the Runtime Infrastructure (RTI) [2, 17].

   For network simulation, PDNS is abbreviated as the Parallel and Distributed Network Simulator [2]. Network Simulator (currently NS2) [21] is the most popular simulation software for telecommunication and networking research. To parallelise NS2, PDNS federates separate NS2 instantiations of different subnetwork modelling on multiple processors. PDNS uses conservative block-based synchronisation implemented in libSynk and RTIKIT. libSynk supports the communication and synchronisation API that scalability can achieve up to fifteen

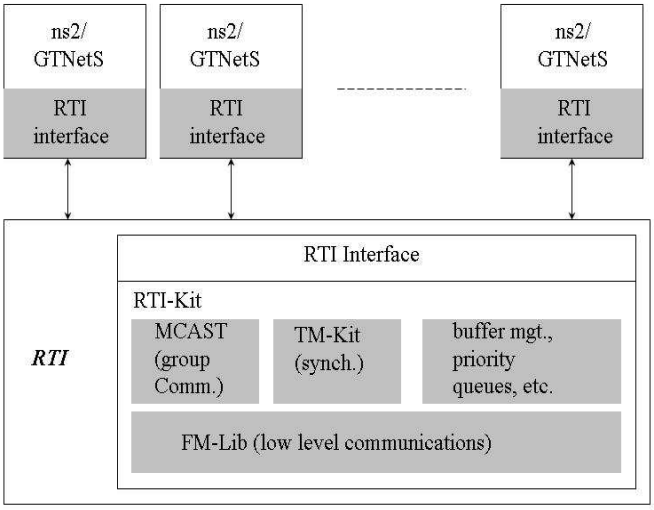

Figure 2.2: PDNS/GTNetS using RTI Library

hundreds processors, whereas RTIKIT is the runtime infrastructure development kit to integrate federated simulation at runtime [2].

Figure 2.2 shows an example of how the RTI library can be interfaced with different simulators, such as PDNS [2] and GTNetS [74]. Fujimoto et al., [69] sees conservative synchronisation to be more suitable than optimistic synchronisation in such federation simulation. Because roll-back required by optimistic synchronisation needs to be implemented in all federated simulators which is against the principle of federation simulation.

4. **Web-based Simulation**

   Web-based simulation runs simulation on the Internet [76]. Such browser-based simulation associates with a world of hypermedia. Education

and Internet gaming are fundamental applications. Common PDES requirements include interactivity and collaborativity. In order to use simulation models distributed on the web, Java and associated platforms become a dominant technology for such web-based simulation, because the object-oriented feature of Java programming language is very suitable for simulation modeling [77].

2.1.8 Performance Studies

Due to the multifaceted diversity of PDES, several aspects, either quantitative or qualitative, are considered important for analysing the performance of PDES. Some quantitative aspects commonly discussed include speedup, processor utilisation, memory utilisation, critical path, and scalability [5, 13, 15, 56, 69].

Speedup measures the objectives of the PDES. Processor utilisation is concerned with how the chosen parallel platform supports the PDES objectives. Memory utilisation shows how well the parallelly available memory resources assist the PDES. Critical path analysis examines the efficiency of different synchronisation protocols. Scalability covers the effectiveness of large scale PDES.

Apart from the quantitative aspects, Reynolds [14] discusses several qualitative issues of design space concerning performance evaluation of PDES.

1. **Speedup**

   - **Simulation speedup** is defined by the simulation time of one processor divided by the simulation time of multiple processors based on the same model. The efficiency of distributed simulation can be derived from such simulation speedup that divides the number of processors and describes how effective the **processor**

utilisation is. The majority of PDES performance analysis is focused on this. Linear speedup will be ideal, however, distributed simulation conventionally needs to pay significant synchronisation overheads [13, 15].

- **Statistical speedup** refers to the simulation time of one processor divided by the simulation time of multiple processors based on the time to obtain system estimates of the same statistical error [13, 15]. Quantitative analysis on PDES, such as the MRIP approach, concentrates on statistical speedup [5].

2. **Memory Utilisation**

   Memory optimality evaluates how well the memory usage can be managed to complete the simulation with the same amount of memory in both parallel and sequential executions. Memory utilisation has a direct impact on PDES using optimistic synchronisation [56].

3. **Critical Path**

   Critical path defines the longest chain of causally dependent events and constrains the execution of a model. To analyse the critical path is to identify possible theoretical parallelism inherent from a model. The critical path can be found either theoretically or in a pre-processing experiment. The results of critical path analysis are important for refining a PDES, especially for model partitioning and load balancing. The experimental results are more realistic than the theoretical ones, especially in new application domains.

   Jefferson and Reiher [78] mention supercritical speedup in which none conservative synchronisation can support parallelism beyond the result of critical path analysis, however, two optimistic synchronisation

approaches are possible to achieve more parallelism than the result of critical path analysis.

4. **Scalability**

   Large-scale simulation, such as the large scale network simulation of the Internet, requires scalable PDES instead of PDES with limited size and complexity. Fujimoto et al., [69] discusses that available memory and simulation time each run are key limitations concerning the sheer volume of packets in large-scale simulation. To evaluate the scalability of a PDES, the number of packet transmissions processed per second of wallclock time (PTS) is an important measure.

   Nicol [79] discusses that scalability may be possible if the increases of model size do not affect load balancing and do not overgrow communication overheads on parallel processors. In general, how scalable a PDES can be mostly depends on how well the trade-off decisions are made between load balancing and synchronisation overhead. A conservative synchronisation model, QS, is implemented to verify such scalability.

5. **Design Space**

   Reynolds established the SPECTRUM testbed (Simulation Protocol Evaluation on a Concurrent Testbed with ReUsable Modules) to support efficient evaluation of PDES. PDES is not all about conservative or optimistic synchronisation approaches. He emphasised a design space of various qualitative aspects of PDES as follows [14].

   - **Partitioning** As a simulation model is partitioned into multiple submodels, the parallelisation of PDES is by distributed submodels or LPs among multiple processors. Partitioning involves iden-

tifying suitable semantic boundaries among distinct sets of simulation states. Such model partitioning and submodels distribution result in local causality constraint that requires synchronising LPs messages either conservatively or optimistically.

- **Adaptability** Optimistic synchronisation may switch to conservative synchronisation subject to the number of rollbacks. For example, dynamic load balancing adapts to changes of workload among LPs during simulation.
- **Aggressiveness** Messages are processed conditionally. Optimistic synchronisation approach, such as time warp, is an example of maximal aggressiveness.
- **Accuracy** After a PDES simulation is finished, the sequential order of distributed message passing is correct. Accuracy is not a requirement, but a definition. Inaccuracy does exist in SRADS [73] and moving time windows.
- **Risk** The purpose of risk is to allow utilisation of otherwise idle computing resources. Either aggressiveness or inaccuracy could lead to initiate or transfer risk messages. How much risk to take is model dependent.
- **Knowledge embedding** Simulation state variables are shared, as semantic attributes of simulation models are used to determine simulation processes. In order to provide useful unconditional knowledge so that the number of non-event messages can be reduced. Usually, knowledge is embedded or state variables are shared via parameter passing at run time. However, knowledge embedding compromises on transparency and flexibility of simulation processes. Whether or not to go embedding knowledge

is an open problem.

- **Knowledge dissemination** An LP sends messages to other LPs in order to allow other LPs to make simulation progress. Examples from conservative protocols include null messages, link time, and appointments. One example from optimistic protocols is an anti-message. Among these, appointment assumes knowledge embedding. Disseminated messages are a form of redundant computation.
- **Knowledge acquisition** LPs request information from the simulation environment on demand in order to make decisions on processing pending event list. Therefore, computation is accountable.
- **Synchrony** Loosely asynchronous, or time based, or timestepped simulation should not be discarded, although the majority of PDES is asynchronous or event based simulation.

2.2 Thoughts on the MDRIP Approach

Stopping rules to determine the simulation run length are implemented in AKAROA2 [1]. The automated data analysers of MRIP in AKAROA2 support for on-line statistical analysis of simulation output data [1, 8]. In developing the MDRIP approach, processes of on-line statistical analysis should be retained and reused to link with distributed simulation. From previous survey of PDES and understanding of MRIP implementation, MDRIP needs to process message passings related to random numbers and observation data. Moreover, it requires a framework of centralised data management, such as the global data analyser, to manage distributed data observation for on-line sequential analysis on simulation output data.

Chapter 3

Design

This chapter starts from overview of MDRIP first, then overviews on MRIP and distributed simulations are also discussed. An introduction on two target models is given with an emphasis on model partitioning and distribution details. Networking architecture is provided based on networking framework, interprocess communication scheme, as well as connection management. System components are presented to summarise features of the new MDRIP functionality.

3.1 Overview of MDRIP

3.1.1 MDRIP

The goal of MDRIP is to run distributed simulation with on-line quantitative and sequential control in multiple replications. In the context of AKAROA2, it is to run distributed simulation under the control of one or more *mdrip engines* centrally managed by the *Global Data Analyser* in *akmaster*. In brief, the purpose of MDRIP is to enable distributed simulation controlled by MRIP.

A conceptual overview of the MDRIP approach is shown in Figure 3.1. The *Global Data Analyser* receives data observation at checkpoints from the *Local Data Analyser* and analysises data according to required statistical accuracy. The *Local Data Analyser* processes different streams of observed data from different submodels.

Different simulation models form different sequencing and causality constraints. How to coordinate these various data streams is dependent on how the semantic relationships among submodels dictate the overall sequencing constraints in the model. Therefore, the logic of such coordinations is different from model to model.

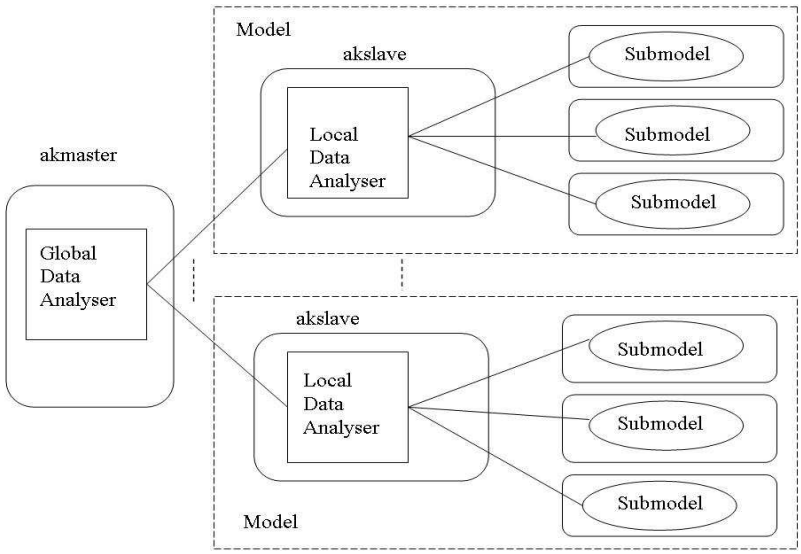

Figure 3.1: Conceptual Overview of MDRIP

Such model-dependent logic for MDRIP can be organised with a composition of multiple linked list queues. Each element in a linked list queue only knows its previous element and its next element. So the data observations messages submitted from a submodel to an *akslave* machine are stored into

an associated linked list queue with sequential ordering preserved. In addition, both insertion and retrieval of data observations from the linked list are very efficient with $O(1)$. Such sequential order preserving is crucial to the composition. Otherwise, those distributed data observations might need to be reordered when received by the *Local Data Analyser*.

*Subengine*s representing subevents from submodels produce estimates. Estimates are data observations. These estimates are sent to the *Local Data Analyser* and are implemented as incoming messages received by the *akslave* which extends *engine* into *mdrip engine* and stores streams of incoming messages into linked list queues accordingly. Because each stream of the incoming messages represents timestamped subevents executed on each *subengine* for each submodels, such incoming messages are sequentially ordered data. Therefore, the first element of each linked list queue is the smallest timestamped subevent from a particular submodel. The *mdrip engine* only needs to repeatedly remove the head of each linked list queue to compose the combined observation data with proper sequencing semantics.

The goal of MDRIP is to combine both MRIP and distributed simulations. Therefore, statistical speedup with adjustable accuracy level can be used to evaluate multiple replications of distributed simulation.

Another view of MDRIP, taking required message passings into accounts, is depicted in Figure 3.2. R_{1xi} stands for the i-th random number for submodel x in the first simulation run. O_{1xi} stands for the i-th data observation of submodel x in the first simulation run.

In the context of AKAROA2, each simulation *engine* is associated with a *Local Data Analyser* which interacts with the *Global Data Analyser* to process quantitative and sequential control. To combine MRIP and distributed simulation, the simulation *engine* is extended to *mdrip engine* which coop-

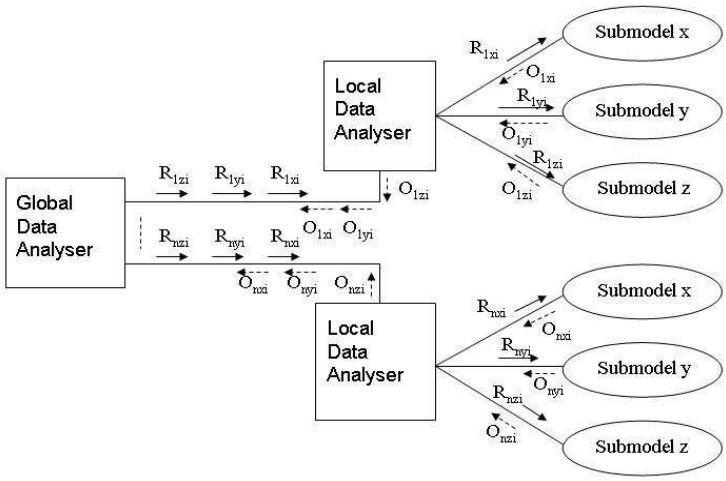

Figure 3.2: Use of Independent Sequences of PRNs in MDRIP

erates with *subengines* of partitioned and distributed submodels executed on multiple processors.

The *mdrip engine* involves passing messages of random numbers and data observations between *subengine*s. MDRIP uses the functionality of stochastic quantitative and sequential control of MRIP which is shown as the message passings between the *Global Data Analyser* and the *Local Data Analyser*. MDRIP adds new functionality between the *Local Data Analyser* and submodels which introduces distributed data observations produced by distributed *subengine*s. Therefore, MDRIP is capable of processing distributed simulation.

3.1.2 MRIP

The MRIP approach currently featured in AKAROA2 supports only non-distributed simulation. A conceptual overview of MRIP is depicted in Figure 3.3. Message passing in the MRIP does not need to be synchronised, because each replication runs a whole simulation model with a stream of independent random numbers. The *Local Data Analyser* does not need to handle different data streams of data observation. There is only one data stream of observed data from *engine* for each *Local Data Analyser*, thus no complicated coordination is required.

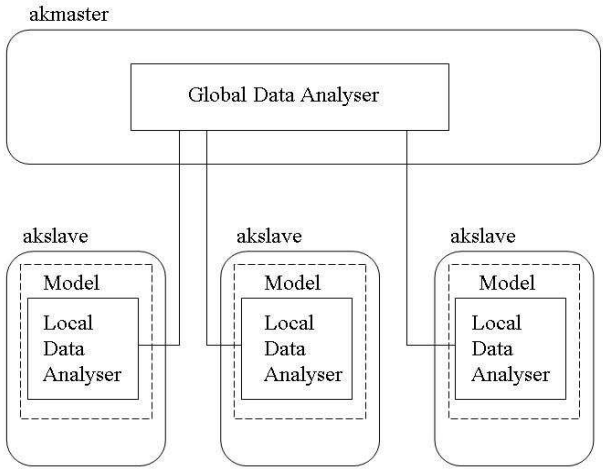

Figure 3.3: Conceptual Overview of MRIP

Messages of random numbers and data observations are sent and received between the *Global Data Analyser* of *akmaster* and the *Local Data Analyser* of *engine* launched by *akslave*. MRIP is the same as MDRIP that interaction between the *Global Data Analyser* and the *Local Data Analyser* comprises

of quantitative and sequential control of on-line stochastic simulation. In comparison of Figure 3.4 and Figure 3.2 shows that MRIP is different from MDRIP in that random numbers and data observations of MRIP are not distributed.

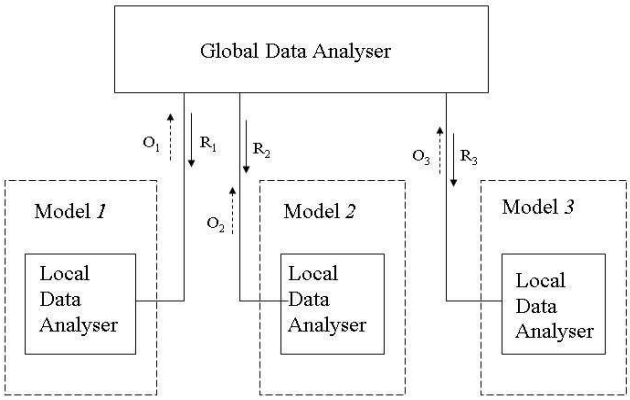

Figure 3.4: Use of Independent Sequences of PRNs in MRIP

MRIP offers statistical speedup because the numbers of observations collected per unit time are proportionally increased as the number of processor is added. However, the MRIP approach is still limited to simulating small and basic models. In addition to limited modeling, the global context of centralised quantitative and sequential control may require huge memory [5, 6].

3.1.3 Distributed Simulations

Most PDES studies [17, 18, 19, 20] are concerned with how to partition a simulation model into submodels and how to distribute submodels across multiple processors and do not pay much attention in statistical accuracy.

Most PDES emphasise on system speedup measured either by simulation run length over number of processors or by processor utilisation. System speedup mainly comes from breaking one very long event list into several smaller subevent lists running among multiple processors. Despite being capable of simulating large and complex simulation models, the synchronisation issues are not easy tasks and need to be well managed.

In the context of AKAROA2, a conceptual overview of a distributed simulation is depicted in Figure 3.5. Various submodels execute on multiple computers and generate distributed observation data.

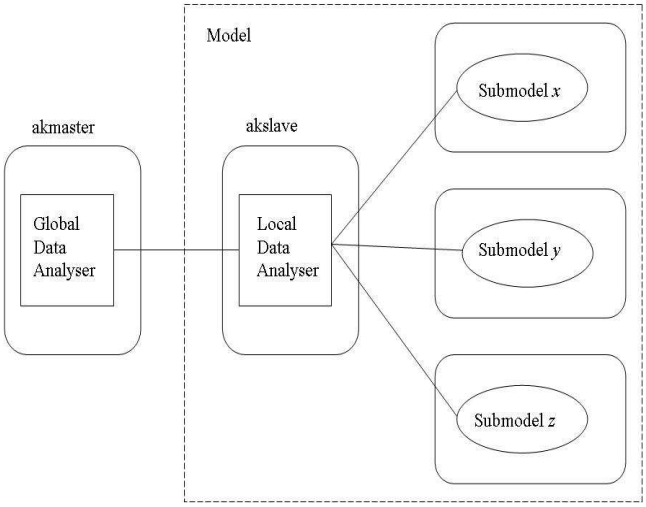

Figure 3.5: Distributed Simulation in AKAROA2

Wu et al., [19] reports an example of a distributed simulation depicted in Figure 3.6. This PDES research relates to parallelisation of the commercial OPNET. Each submodel is equivalent to each sequential simulator as a federate. The whole simulation model composed from the model repository acts as a federation. Each submodel runs on one computer and exchanges data between different submodels on other computers by message passings. RTI library performs the necessary synchronisation among subevents of different submodels according to certain synchronisation protocols. In this case, the source code of the OPNET simulator is not available, therefore, a proxy function is arranged to transform required message formats between each two sequential simulators regardless of whether they are the same simulators or not.

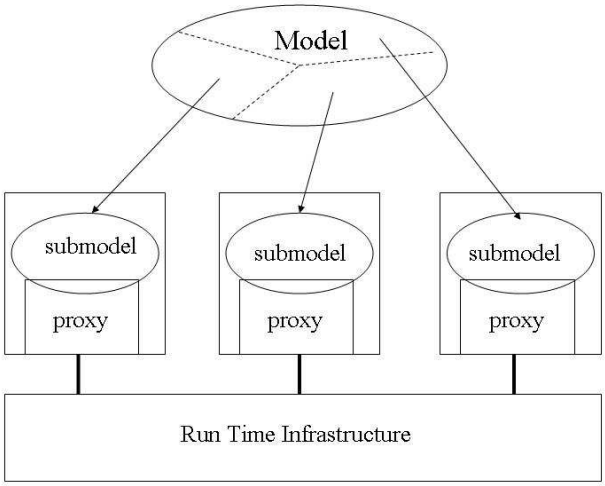

Figure 3.6: A Model Partitioned and Distributed

Riley et al., [18] reports another example of a distributed simulation as shown in Figure 3.7. It describes a generic framework for parallelisation of different sequential simulators, such as OPNET and NS. Because different simulators have different message formats, similar to the proxy function in [19], a generic Interface is implemented to transform message formats between different sequential simulators when RTI performs synchronisation of distributed events from different simulators.

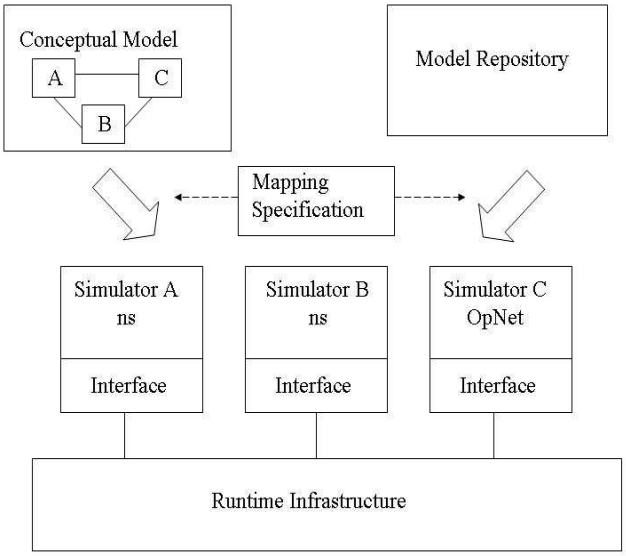

Figure 3.7: Distributed Simulation with Different Simulators

A Conceptual Model as a whole model in the upper-left box of Figure 3.7 demonstrates a distributed simulation consisting of Simulator A, Simulator B, and Simulator C. Each sequential simulator represents a submodel of the whole model which is composed from the model repository. This case experiments interoperability between sequential simulators and reuse of different

model implementation. From the point of view of federated simulation, each sequential simulator acts as a federate and the whole model is a federation.

Upon three cases of distributed simulation discussed, the one in the context of AKAROA2 is different from the other two supported by RTI library. The two major differences are assumption of on-line sequential control of statistical errors and possibility of causality constraints.

The two distributed simulations supported by RTI library do not assume that statistical errors have to be sequentially controlled, while the distributed simulation in AKAROA2 is restricted to on-line sequential control of statistical errors. The advantages of assuming on-line sequential control of statistical errors have been explained in Section 2.1.6. The disadvantage of assuming on-line sequential control of statistical errors of final results is that it can require huge amount of global data to keep track of all checkpoints of data analysis.

The two distributed simulations supported by RTI library are capable of dealing with possible causality constraints, because communication and synchronisation services have been supported by RTI library through Interface [18] or Proxy [19]. However, the distributed simulation in AKAROA2 has not yet been examined by causality constraints. It needs to be pointed out that communication and synchronisation services provided by RTI library are not automatically and seamlessly applicable. If a simulator needs to communicate with the other simulator through RTI library, then both simulators need to implement their own Interface [18] or Proxy [19] functions.

The development of MDRIP at this stage assumes only the on-line sequential control of statistical errors of final results. Support for causality constraints has not yet been included and will be model dependent.

3.2 Modeling Phase

3.2.1 The Target Models

This thesis considers two queueing network models. The first one is a queueing network with four independent queueing systems, as shown in Figure 3.8. The second one is a queueing network with tandem connection of two servers, as shown in Figure 3.9.

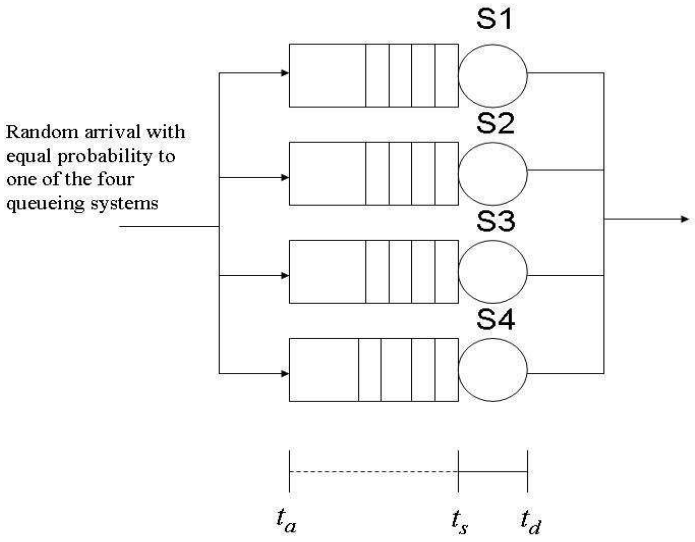

Figure 3.8: First Queueing Network - Four Independent Queueing Systems

In the first queueing network, customers randomly arrive in the queueing network and are dispatched to one of the four independent queueing systems S1, S2, S3, or S4 with equal probability. The services and service rates provided by these four independent queueing systems are assumed to be the same. Once a service is completed, the customer leaves the queueing network.

t_a denotes when a customer arrives in the queueing network for either S1, S2, S3, or S4 queueing systems. t_s denotes when a customer gets served

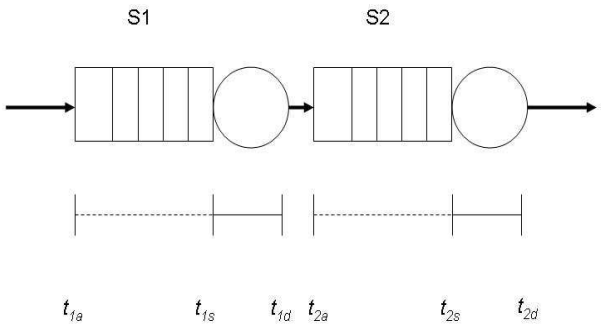

Figure 3.9: Second Queueing Network - Tandem Connection

by one of the four queueing systems. t_d denotes when a customer departs from the service of the queueing network.

In the second queueing network, customers randomly arrive in the first queue, wait for the service from the first server (S1), get served by S1, head to the second server (S2), wait for the service from S2, get served by S2, then leave this tandem queueing network.

t_{1a} denotes timestamp when a customer arrives in S1. t_{1s} denotes when a customer gets served by S1. t_{1d} denotes when a customer departs from the service of S1. t_{2a} denotes when a customer arrives in S2. t_{2s} denotes when a customer gets served by S2. t_{2d} denotes when a customer departs from the service of S2.

To keep model related issues in a consistent context, the following definitions and formula [62] are briefly explained:

- ρ = server utilisation
- μ = traffic intensity
- λ = arrival rate
- T_S = the mean service time
- T_W = the mean waiting time
- T_Q = the mean response time
- $\rho = \lambda T_S$
- $T_W = \frac{\rho T_S}{1-\rho}$
- $T_Q = T_W + T_S$

For the first queueing network, $(t_s - t_a)$ or T_W defines the mean waiting time value of how long a customer needs to wait for the service of one of the four queueing systems, S1, S2, S3, or S4. $(t_d - t_s)$ or T_S gives the mean service time for how long a customer spent in service by one of the four queueing systems. $(t_d - t_a)$ or T_Q tells the mean response time for how long a customer stays in the queueing network.

For the second queueing network, $(t_{1s} - t_{1a})$ or T_{1W} gives the mean waiting time value of how long a customer needs to wait for the service of S1. $(t_{1d} - t_{1s})$ or T_{1S} tells the mean service time for how long a customer gets served by S1. $(t_{1d} - t_{1a})$ or T_{1Q} yields how long a customer spends in the first queue and is often described as the mean response time. Accordingly, $(t_{2s} - t_{2a})$ or T_{2W} for the mean waiting time in S2, $(t_{2d} - t_{2s})$ or T_{2S} for the mean service time in S2, and $(t_{2d} - t_{2a})$ or T_{2Q} for the mean delay or response time in S2. The total mean response time that a customer spends

in this queueing network consists of the mean response time in S1 and the mean response time in S2, $(T_{1Q} + T_{2Q})$ or simply $(t_{2d} - t_{1a})$.

The statistics of interests in this thesis are concentrated on the total mean waiting time, the total mean service time, and the total mean response time of the queueing networks. For the first queueing network, the total mean waiting time is simply T_W, the total mean service time is T_S, and the total mean response time is T_Q. For the second queueing network, the total mean waiting time is $(T_{1W} + T_{2W})$, the total mean service time is $(T_{1S} + T_{2S})$, and the total mean response time is $(T_{1Q} + T_{2Q})$.

3.2.2 Partitioned and Distributed Models

The ideas of non-partitioned and non-distributed simulations of the target models run by one *engine* on one processor are shown in Figure 3.10 and Figure 3.11.

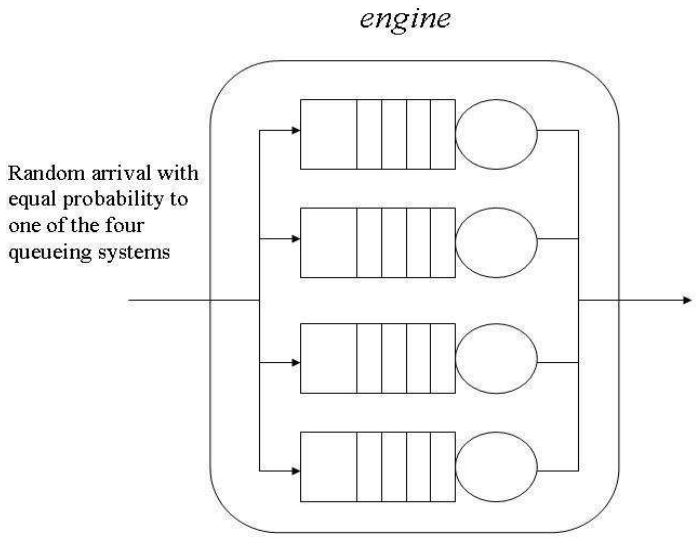

Figure 3.10: Non-partitioned and Non-distributed - First Queueing Network

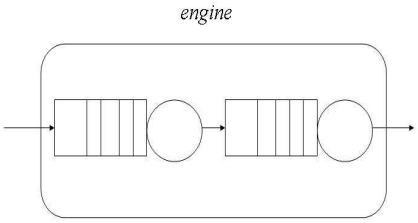

Figure 3.11: Non-partitioned and Non-distributed - Second Queueing Network

In general, each round-edged square box represents one processor. The arrow lines represent data flows that describe the semantic relationship which forms the sequencing constraint of the target models.

For the first target model, the semantic information describes that submodels w, x, y, and z happen with equal probability. There is no sequence in submodels, therefore, there is no sequencing constraint in this model. Customers enter into the system are served by one of the four servers with equal probability.

For the second target model, the semantic information describes that submodel x must happen before submodel y, because customers depart from the first server simulated by submodel x and arrive in the second server simulated by submodel y. Customers do not depart from the second server and then arrive the first server. Customers do not leave the second server and go back to arrive in the first server, either. Therefore, the sequencing

constraint dedicates that each subevent from the *subengine x* needs to be followed by a subevent from the *subengine y* to complete an event with an observed data composed for reporting to the *Global Data Analyser* in *akmaster*.

The ideas of partitioned and distributed models run by *subengine*s on multiple processors are shown in Figure 3.12 and Figure 3.13.

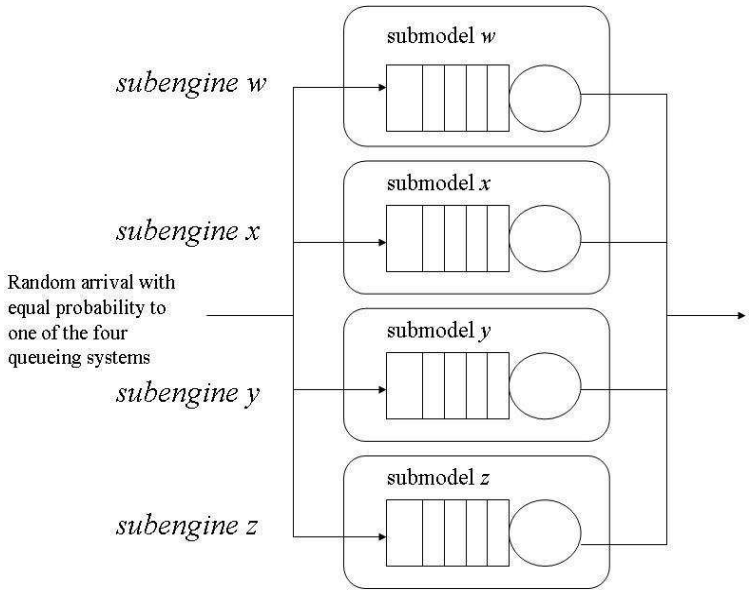

Figure 3.12: Partitioned and Distributed - First Queueing Network

In Figure 3.13, the thick black line between *subengine x* and *subengine y* indicates distributed messages directly passing between two networked processors.

In Figure 3.12, because there is no sequencing constraint, there is no thick black line in between *subengines w, x, y, and z.* Data exchanged directly between *subengine*s reflect the semantics of the model among distributed submodels. In the case of the tandem queueing network shown in Figure 3.13, data flown from the first *subengine* to the second *subengine*

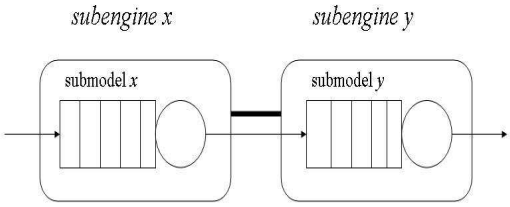

Figure 3.13: Partitioned and Distributed - Second Queueing Network

represents subevents happened in the first queue are followed by subevents happened in the second queue.

Referring to Figure 3.11 and Figure 3.13, it is observed that such queueing network in tandem does not include any semantic cycle, because only two nodes (submodel x and submodel y) and one directed link from the first node (submodel x) to the second node (submodel y). Therefore, the consequent model partitioning does not expect any causality constraints. As a result, the composition of distributed data observation in the *mdrip engine* for such a tandem queueing network is simply an addition.

In Figure 3.13, the *mdrip engine* expects to receive two data streams of data observation. One is from the *subengine x* simulating the first queueing system, the other is from the *subengine y* simulating the second queueing system. Because two queues connected in tandem dictate that data from the

first data stream should always be followed by data from the second data stream accordingly. Therefore, the coordination is simply iterative additions of the according data from one data stream and the other that follows.

Further implementation and testing can see that the *mdrip engine* supports both target models with on-line sequential control of statistical errors. The first target model, in Figure 3.8, with four independent queueing systems should produce the same estimates results whether such model is run non-partitioned and non-distributed on one processor or is run partitioned and distributed on four processors. The second target model, in Figure 3.9, with two tandem connected queueing systems should produce similar estimates results whether such model is run non-partitioned and non-distributed on one processor or is run partitioned and distributed on two processors.

3.3 Networking Architecture

3.3.1 Server/Client Framework

The framework of server/client or master/slave is the communication pattern used by MRIP in AKAROA2 [1, 4].

With regards to reusability and extensibility, the *server side* and the *client side* are the same as under the MRIP approach. The *server&client side* is the new element designed to fit into the existing MRIP. The purpose of this new networking element is to intercept existing sequential control messages of MRIP and intercommunicate these messages with the distributed *subengines*.

The overall message passings are intercommunicated among three parts of the networking architecture. The following pseudo code describe how file descriptors are structured to facilitate the networking architecture for the MDRIP.

- server side

```
socket listen_file_descriptor
bind listen_file_descriptor
listen listen_file_descriptor
for loop
    select listen_file_descriptor
    if client requests a connection
        accept listen_file_descriptor as connection_file_descriptor
    for loop
        request from client
        respond to client
    end loop
end loop
```

- server&client

```
socket socket_file_descriptor
connect socket_file_descriptor
socket listen_file_descriptor
bind listen_file_descriptor
listen listen_file_descriptor
for loop
    select listen_file_descriptor
    if client in client array requests a connection
        accept listen_file_descriptor as connection_file_descriptor
    for loop
        select listen_file_descriptor
           request from client
           respond to client
    end loop
end loop
request to server
respond from server
```

- client side

```
socket socket_file_descriptor
connect socket_file_descriptor
request to server
respond from server
```

3.3.2 I/O Multiplexing

I/O multiplexing with the `select` function is used to support the multitasking of networking architecture for the MDRIP approach. I/O multiplexing is one of the five I/O models available under UNIX [27]. The reasons to use I/O multiplexing include:

- reusability and extensibility of MRIP in AKAROA2,
- the *client side* and the *server&client side*, each needs to handle multiple file descriptors and multiple sockets at the same time,
- the *server side* and the *server&client side*, each needs to handle a listening socket and its connected sockets,
- multiple protocols are possible.

The MDRIP approach requires two-stage I/O multiplexing. The multiply of replications in MRIP between one *akmaster* and several *akslaves* are intercommunicated by the first stage I/O multiplexing. The *akslave* launches several simulation *engines* by using `fork` method to create different child processes. MDRIP requires the extended *mdrip engine* to act as an intermediate agent that coordinates random number requests and allocations between *subengines* and *akmaster* as well as collects observed data from *subengines* via the *Local Data Analyser* to the *Global Data Analyser* in *akmaster*. The *mdrip engine* intercommunicates with multiple distributed *subengines* through the second stage I/O multiplexing.

Referring to the networking architecture of the MDRIP, the first stage I/O multiplexing corresponds to the intercommunication between the *server side* and the *server&client side*. The second stage I/O multiplexing corresponds to the intercommunication between the *server&client side* and the *client side*.

3.3.3 Connection Management

The concept of '*publish and subscribe*' is used to manage the networking connection of server/client (or master/slave) framework in between *akmaster* and *akslave* as well as between *mdrip engine* and *subengine*.

In MRIP, the *server side* publishes its network address, while the *client side* retrieves the network address of the *server side* and subscribes to its services. In extension to the MDRIP, the *server side* publishes its network address, while the *server&client side* retrieves the network address of the *server side* and subscribes to its services. At the same time, the *server&client side* publishes its network address as well, while the *client side* retrieves the network address of the *server&client side* and subscribes to its services.

In AKAROA2, *akmaster* publishes and stores its network address into the `./akmaster` file in which *akslave* looks for values of networking address and connects to *akmaster* process. In the MDRIP extension, the *mdrip engine* publishes and stores its network address into the `./sripslave` file. The *subengine* processes look for the network address of the related *mdrip engine* and connects to that process.

The '*publish and subscribe*' concept used in connection management provides flexibility in building communication topology. Such flexibility is useful in PDES implementation. Easy maintenance can be expected for large size model with more networked processors. Separation of communication flows and observation data flows is advantageous for future interoperability.

3.4 System Components

The following details of system components reflect the process-oriented designs from MRIP to MDRIP. Figures in this section are notated with a round-edged rectangle representing a physical processor, a rectangle box representing each UNIX process. Letters in box describe names of processes. Lines between rectangle boxes represent messages being passed around different UNIX processes.

Since the design of MDRIP is to extend distributed simulation functions from MRIP, reusability and extensibility are highly regarded as very important qualities.

In AKAROA2, functions related to sequential control of statistical accuracy are very crucial software modules. Tremendous time and efforts have been dedicated to design, implement, test, and validate these modules in AKAROA2 [1, 3, 4, 5, 6, 7]. Effective reuse of these modules will shorten the time needed to implement the MDRIP. Therefore, it is necessary to examine the existing MRIP to verify such reusability practice.

If such reuse is effective for implementing the MDRIP approach, then the automation of sequential data analysers in AKAROA2 will be more transparent for further extensions to the other PDES simulators other than within the AKAROA2.

3.4.1 The Existing MRIP

The existing MRIP approach under AKAROA2 features networking architecture of the *server side* and the *client side*, see Figure 3.14. The `akmater` process launches `akslave` processes. The `akrun` process launches the `simulation` process that asks the `akmaster` process to instruct the `akslave` process to `fork` another unix process to launch simulation `engine` processes. An `engine` process utilises processor cycles of a *client side* machine to process the event list of a simulation model and also connects with the *server side* machine to report observation data back to the `akmaster` process for further analysis and report preparation. More than one `akslave` process can be launched and each `akslave` process can `fork` more than one `engine` process.

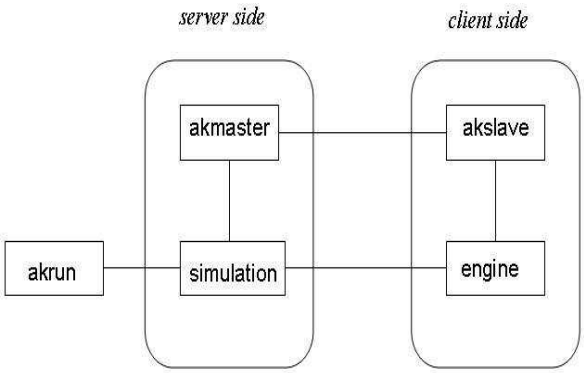

Figure 3.14: *engine* in MRIP

3.4.2 The New MDRIP

The existing MRIP is extended into the new MDRIP. The new MDRIP approach is structured into the networking architecture of *server side*, intermediate *server&client*, and *client side*. `akmaster`, `akslave`, `akrun`, and `simulation` processes all remain unchanged which has been described in the previous subsection. Only the `engine` process is modified to include MDRIP functions that transform the previous *client side* into an intermediate *client&server*. Such a *mdrip engine* process coordinates message passings between *server side* and *client side*s. Distributed *subengine*s run on *client side*s and utilise their processors' cycle to process each sub-eventlists. Figure 3.15 demonstrates the new MDRIP extended from MRIP and consisting of *mdrip engine* and distributed *subengine*s.

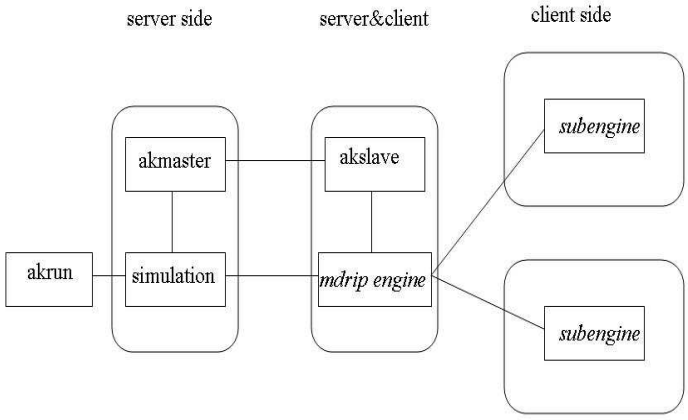

Figure 3.15: *mdrip engine* and *subengine*s in MDRIP

3.4.3 Publish and Subscribe

Features underlining the connection management which are extended from MRIP to MDRIP are shown in Figure 3.16. Three new processes are created: *subengine*, *sripslavemaster_to_client*, and *client_to_sripslavemaster*. The *server side* does not need to know the address of the *server&client side* beforehand. The *client&server side* does not need to know the address of the *client side*, either. Instead, the *server side* publishes its address by getting address information from environment variables and writing it into a specified file. So does the *server&client side*. When a *mdrip engine* is initiated, it searches for that specified file and reads the address information of the *akmaster*, then subscribes to the *server side* by the `connect` method in UNIX. When a *subengine* is initiated, it searches for that specified file and reads the address information of the *mdrip engine*, then subscribes to the *server&client side* by the `connect` method.

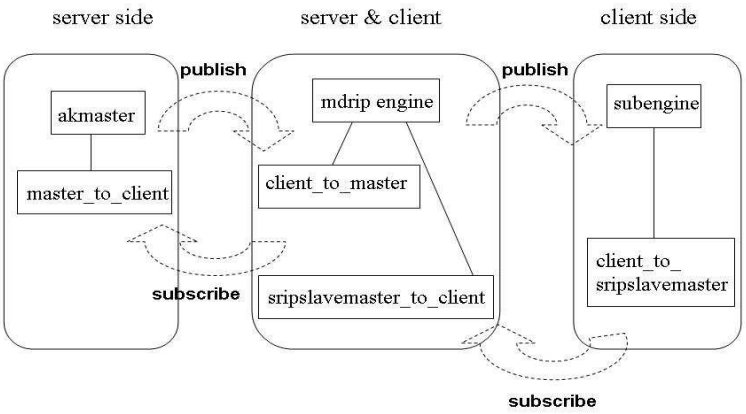

Figure 3.16: Connection Management

3.5 Summary

This chapter outlines the design of the MDRIP approach with overviews on MDRIP, MRIP, and distributed simulations. As MDRIP is extended from MRIP, several conceptual overviews show how the *Global Data Analyser* and the *Local Data Analyser* interact with multiple submodels which are run by *subengine*s. Two cases of distributed simulation with utilisation of RTI library and without on-line output data analysis are also discussed.

Two target models based on queueing systems are analysed for their system behaviours. Model partitionings are performed and submodels are distributed.

Networking architecture of MDRIP is discussed with the framework of server/client relationship, the I/O multiplexing in interprocess communication, and the connection management for flexible communication topology..

To progress from the conceptual design to implementation, system components are illustrated from the existing MRIP to the new MDRIP, as well as the '*publish and subscribe*' connection management.

Chapter 4

Implementation

The MDRIP implementation is based on UNIX socket programming in a Linux environment networked over the TCP/IP protocol. To ensure compatibility with the existing MRIP approach in AKAROA2, the C/C++ programming language is used. Message formats are followed. The TCP/IP protocol is chosen for reliable packet transmission needed by sequential control of statistical output data analysis in the MDRIP approach.

4.1 Sequential Control Messages from MRIP to MDRIP

- Important sequential control messages in MRIP include:

  M_RNDQ to request random numbers

  M_RNDA to allocate random numbers

  M_CKPT to contain observed data at each checkpoint

  Table 4.1 illustrates the message passing relationship between M_RNDQ, M_RNDA, and M_CKPT messages. The basic source of random numbers in AKAROA2 is `AkRandomReal()` that requires M_RNDQ and M_RNDA messages. Currently, `AkRandomReal()` uses a Combined

Multiple Recursive pseudo random number generator (CMRG) with a cycle period of around 2^{191} [1]. The `AkObservation` is the interface routine between simulation modeling program (simulation engine) and the data analysers. This interface routine requires M_CKPT messages.

Table 4.1: Message Passings: M_RNDQ, M_RNDA, and M_CKPT

server side	client side
	$<-$ send M_RNDQ
receive M_RNDQ $<-$	
send M_RNDA $->$	
	$->$ receive M_RNDA
	$<-$ send M_CKPT
receive M_CKPT $<-$	

- Important sequential control messages in MDRIP include:

  M_RNDQ to request random numbers

  M_RNDA to allocate random numbers

  M_OBSV to contain timestamped data observation

  M_CKPT to contain observed data at each checkpoint

  M_OBSV is the new message created to coordinate distributed features for MDRIP. Table 4.2 illustrates the relationship of message passing between M_RNDQ, M_RNDA, M_OBSV, and M_CKPT messages. The basic source of random numbers in AKAROA2 is still `AkRandomReal()` that requires the same format of M_RNDQ and M_RNDA messages. However, such relationship is modified in that a *server&client* part is added to intercept and relay M_RNDQ and M_RNDA messages. Partitioned submodels are run on distributed

*subengine*s so that each *subengine* sends observed data via the new `AkSripObservation` interface routine. This `AkSripObservation` interface routine sends M_OBSV messages to the intermediate *server&client* part where *mdrip engine* receives M_OBSV messages, coordinates distributed features, and uses `AkObservation` interface routine to send distributed observation for analysis on statistical errors.

Table 4.2: Message Passings: M_RNDQ, M_RNDA, M_OBSV and M_CKPT

server side	server&client	client side
		$<-$ send M_RNDQ
	receive M_RNDQ $<-$	
	$<-$ send M_RNDQ	
receive M_RNDQ $<-$		
send M_RNDA $->$		
	$->$ receive M_RNDA	
	send M_RNDA $->$	
		$->$ receive M_RNDA
		$<-$ send M_OBSV
	receive M_OBSV $<-$	
	call `AkObservation` routine	
	$<-$ send M_OBSV	
	$<-$ send M_CKPT	
receive M_CKPT $<-$		

4.2 The *subengine*

In MDRIP, one or many *subengine* processes run partitioned and distributed submodels as a simulation program. The objective of a *subengine* process is to process a shorter sub-event list. The message passing of a *subengine* is mainly involved at the *client* side. Table 4.3 shows that a *subengine* requests and obtains random numbers for running its partitioned submodel and calling `AkSripObservation` routine to report its observation data.

Table 4.3: Message Passings for *subengine*

server side	server&client	client side
		$<-$ send M_RNDQ
		$->$ receive M_RNDA
		call `AkSripObservation` routine

Unlike the MRIP approach where a simulation program is run as a simulation *engine* launched by *akslave* process, the simulation program needs to include `akaroa.H` header file for necessary processes with *akmaster*, such as sending back observations via `AkObservation` routine. To get random number streams from *akmaster*, the simulation program includes `akaroa/distributions.H` header. To use some basic modeling constructs, inclusion of `akaroa/process.H` provides `Process` class and `Hold` for blocking the current process for a given amount of simulation time. To include `akaroa/resource.H`, `Resource` class can `Acquire`, `Release`, or `Remove`.

The same as *engine* in MRIP, the *subengine* for MDRIP includes `akaroa.H`, `akaroa/distributions.H`, and/or `akaroa/process.H` and `akaroa/resource.H`. The *subengine* process does produce observation data, but it send the observation data to the `AkSripObservation` routine instead of the `AkObservation` routine. Thus, the inclusion of `AkSripObservation.H` is necessary.

4.3 `AkSripObservation` routine

To extend MRIP to MDRIP, an `AkSripObservation` routine is created to intermediate between simulation *subengines* and *engines*.

`AkObservation` is the most important AKAROA2 library routine. It takes an observation and interfaces it with *akmaster*, updating associated estimates until the required accuracy is reached [1]. The use of `AkObservation` realises the automation of stochastic quantitative and sequential control on simulation output data. It is crucial for `AkSripObservation` routine to reuse or be compatible with `AkObservation`.

`AkSripObservation` takes an observation from a *subengine* and sends such observation to the *mdrip engine* that collects and coordinates distributed observations from distributed *subengines*. The message passing related to the `AkSripObservaton` routine is shown in Table 4.4.

Table 4.4: Message Passings for `AkSripObservation`

server side	server&client	client side
		$<-$ send M_OBSV

Once the `AkSripObservation` routine is called, it calls the modified `GetMasterConnection` that establishes a connection between the *subengine* process and the *engine* process to pass observation data from *subengine* to *engine*. Thus, it is necessary to include `sripslavemaster_to_client.H`, `client_to_sripslavemaster.H`, and
`../engine/engine_to_master.H`.

Once the *mdrip engine* receives the M_OBSV messages sent by *subengine*s calling `AkSripObservation` routine, the *mdrip engine* processes distributed M_OBSV messages, calls `AkObservation` routine, and passes in the processed values as the parameters.

4.4 `sripslavemaster_to_client` and `client_to_sripslavemaster` routines

The features of `sripslavemaster_to_client` and `client_to_sripslavemaster` routines enable the connection management between the *mdrip engine* and the *subengines*. It is similar to the interaction of connection management between the *akmaster* and the *slaves*.

`sripslavemaster_to_client` is used by the *mdrip engine* to publish the host address where the *mdrip engine* is initialised. The value of such host address is written into a hidden file called ".sripslave". It is almost identical for `master_to_client` that *akmaster* publishes the host address where the *akmaster* process is initialised. The difference is that the host address of *akmaster* is written into a hidden file named ".akmaster".

`client_to_sripslavemaster` is used by processes that need to subscribe to the connection to the process of *mdrip engine*. It is almost identical to `client_to_master` used by the processes to subscribe to the connection to the *akmaster* process. The difference is that `client_to_sripslavemaster` arranges a process to connect to the *mdrip engine* by the host address information published by `sripslavemaster_to_client` in the hidden file "sripslave", while `client_to_master` arranges a process to connect to the *akmaster* by the host address information published by `master_to_client`.

Inclusions of `sripslavemaster_to_client.H` and `client_to_sripslavemaster.H` are necessary for use of these two routines.

4.5 `GetMasterConnection` routine

The `GetMasterConnection` routine in `engine_to_master.C` takes no parameter and returns an object of `Connection` class. Whenever a process calls this routine, a connection to a target process is established and ready for massages passing over it. The target process for the original `GetMasterConnection` is the *akmaster* process. Therefore, all the processes that call the original `GetMasterConnection` routine get connected to the *akmaster* process and are able to send and receive messages with the *akmaster* process.

Since the design of the MRIP approach in AKAROA2 has laid tremendous responsibility on the *akmaster* process, the `GetMasterConnection` routine is very frequently called. It is an obviously better practice that the implementation of MDRIP should take the code reuse into account. To follow the design of extending from MRIP to MDRIP and to reduce the implementation time, all the functions related to quantitative and sequential control in the well-tested MRIP approach should be retained.

It is identified that the `GetMasterConnection` is the key routine to be changed. The modification on the `GetMasterConnection` routine focuses on the differentiation between host addresses obtained from the environment variables. The host address of the *akmaster* process is taken as the default value. Once the modified `GetMasterConnection` is called, the called routine first checks if it can read the host address from the ".akmaster" hidden file. If it fails to find the ".akmaster" hidden file, it means that the called routine is seeking the connection to the *mdrip engine* instead of the *akmaster*. Therefore, the connection to be established should be the connection to the *mdrip engine* by the `OpenSripConnection()` from the `client_to_sripslavemaster`. If it is successful to find the ".akmaster" hidden file, then it means that the called routine is seeking the connection to

the *akmaster* instead of the *mdrip engine*. As a result, the connection to the *akmaster* is established by the `OpenConnection(host,post)` from the `client_to_master`.

In brief, the modified `GetMasterConnection` provides the new connection management required for the extension from the MRIP to the MDRIP. In addition, all the other functions that need the original `GetMasterConnection` remain unchanged.

4.6 The *mdrip engine*

The purpose of the *mdrip engine* is to modify the original simulation *engine* managed by *akmaster* and launched by *akslave*. The modifications focus on the initialisation of message passing structure required by the *mdrip engine*, the subscription (or connection) to the *akmaster*, the publishing of the host address to the interested *subengines* and the required message passings and coordinations for new messages: M_RNDQ and M_OBSV. Like the origional *engine*, the *mdrip engine* is still managed by *akmaster* and launched by *akslave*.

The implementation of *mdrip engine* mainly consists of initialising the *mdrip engine*, subscribing and connecting to the *akmaster*, as well as a `Srip()` loop which intercommunicates message passings of random numbers and data observations between *akmaster* and *subengine*s. The message passings related to the *mdrip engine* are shown in Table 4.5.

Table 4.5: Message Passings of *mdrip engine*

server side	server&client	client side
	receive M_RNDQ < −	
	< − send M_RNDQ	
	− > receive M_RNDA	
	send M_RNDA − >	
	receive M_OBSV < −	
	call `AkObservation` routine	
	< − send M_OBSV	
	< − send M_CKPT	

- Initialise the *mdrip engine*

  Following three routines facilitate the initialisation of the *mdrip engine*:

  1. `InitSripTables()`: similar to the `InitTables()` in `akmaster.C` for the storage of related *file descriptors* in an array except that this array is arranged for the message passings on the socket connection between *mdrip engine* and *subengines*.

  2. `InitSripSockets()`: similar to the `InitSockets()` in `akmaster.C` for binding and creating a *listen* socket except that this *listen* socket is created for the connection between *mdrip engine* and *subengines*.

  3. `InitSripMasterAddress()`: similar to the `InitMasterAddress()` in `akmaster.C` except calling `GetSripslaveMasterAddress` in `sripslavemaster_to_client.C` to query if any host address information is available in the ".sripslave" hidden file. If such host address is not available, it means the first initialisation of the *mdrip engine*. If such host address is available, it means more

than once the the mdrip engine has been initialised and indicates the situation of multiple replications.

- Subscribe to the *akmaster*

  After the *mdrip engine* is initialised, a connection to the *akmster* is established by subscribing to the *akmaster* calling the modified `GetMasterConnection()`. Because the *mdrip engine* is launched by the *akslave* on the same host machine, the searching for ".akmaster" hidden file should be successful and a socket `connect` will be subscribed to the *akmaster* according to the host address from the ".akmaster" hidden file.

  This subscription provides socket connections needed between *akmaster* and *mdrip engine* and ensures that all the quantitative and sequential controls under the MRIP approach are maintained under the MDRIP approach.

- The `Srip()` loop

  After the *mdrip engine* is initialised and the connection to the *akmaster* is subscribed, the `Srip()` loop is called to form the second layer of the two-layer I/O multiplexing for intercommunicating message passings between *mdrip engine* and *subengines*.

  - Two-layer I/O multiplexing

    According to the server&client side of the server/client framework specified in the design of networking architecture and the choice of I/O multiplexing used for the MDRIP approach, a two-layer multiplexing is formed.

    Inside the `Srip()` loop features a `for` loop to `select` specified socket connections by an array of *file descriptors* and `accept` in-

coming connections from the *subengines*.

When the `accept` of incoming connections receives a *listen* socket, a new socket connection will be made for a particular `subengine` with a specified *file descriptor*. Further message passings between the *mdrip engine* and that particular *subengine* will utilise such connection. Three possible messages are discussed in details as follows.

– Three possible messages

Once a new socket connection is established, the *mdrip engine* is ready to receive messages from a particular *subengine* and the particular *subengine* is ready to send messages to the *mdrip engine*. Following three messages are possibly expected:

* The M_VREQ message

  This message has nothing to do with the MDRIP approach. It acts as a by-pass action concerning with code reuse of the original MRIP functions.

* The M_RNDQ message

  The original M_RNDQ message format remains unchanged. No new message format is required for the *mdrip engine* and the *subengines* to intercommunicate random numbers. The original *engine* requests and consumes the random numbers, while the *mdrip engine* transfers and relays the random numbers. The *subengine* is similar to the *engine* that only requests and consumes random numbers.

* The M_OBSV message

  This is a new message format that consists of one integer for the number of the parameter and two real values for the

value of observed data and the value of the timestamps. Once a M_OBSV message is working, the associated logic of distributed data observation is processed in order to produce a value composed from a set of distributed LPs.

– Composition of distributed data observation

Considering the tandem queueing network of the second target model in Figure 3.9, the *mdrip engine* needs to arrange two linked list queues to store or enqueue incoming messages from S1 and S2 separately. Once all linked list queues are not empty, then the *mdrip engine* will start to dequeue each linked list queue and compose these dequeued data. For the example of the tandem queueing network, this composition involves the dequeued data of the first linked list queue added onto the dequeued data of the second linked list queue according to the semantics of tandem relationship. If the estimate of interests is the total service time, then each such addition shows the total service time of a customer in the tandem queueing network. If the estimate of interests is the total response time, then each such addition represents the total response time of a customer in the tandem queueing network.

Following code snippet explains the **compose** feature in `Srip()` loop for the second target model. `lr1` and `lr2` stand for linked list one and linked list two. `v1, v2` are the observation values of dequeued data of the incoming subevents. `t1, t2` are the distributed timestamp values of dequeued data of incoming subevents. Values of dequeued data are produced in *subengine*s where submodels are executed. `data1, data2` are data elements in the linked lists. The `v` is the value of data observation after the **compose** feature is processed in that `v = v1 + v2`. The `+`

operation is the **compose** feature which is an addition operation simulating the first queueing system connected with the second queueing system in tandem. Each `v` value is then passed as a parameter of `AkObservation` routine. As a result, on-line sequential control of statistical errors implemented in MRIP is extended to MDRIP.

```
if (lr1 != NULL && lr2 != NULL) {
    v1 = lr1->data1;
    t1 = lr1->data2;
    v2 = lr2->data1;
    t2 = lr2->data2;
    v = v1 + v2;
    lr1 = remove(lr1);
    lr2 = remove(lr2);
    AkObservation(v);
}
```

If the target model is changed, then the **compose** feature needs to be changed to follow the correct semantics of a new target model. The number of linked list queues in *mdrip engine* will need to be changed to reflex the number of multiple processor required by the new target model. For example, the first target model in Figure 3.8 and in Figure 3.12 requires four processors for four *subengine*s to run four submodels. As a result, four linked list queues need to be arranged in the `Srip()` loop for the associated **compose** feature. Details are discussed in 5.2.5 for composing four linked list queues.

Figure 4.1 explains how the linked list queues work in the *mdrip engine* for the second target model. The *mdrip engine* is responsible in composing such data observation and submitting such

composite observed data to the *Local Data Analyser* that will report to the *Global Data Analyser*.

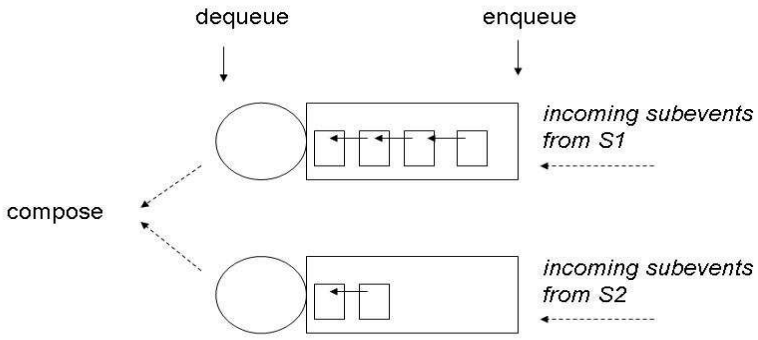

Figure 4.1: Two Streams of Linked List Queue in *mdrip engine*

4.7 Summary

This chapter discusses the implementation of MDRIP. Major features of MDRIP are explained through message passings, related routines, and processes. These features include the sequential control messages from MRIP to MDRIP, the *mdrip engine* process, the *subengine* process, the `AkSripObservation` routine, the `sripslavemaster_to_client` and `client_to_sripslavemaster` routines, and the `GetMasterConnection` routine.

Source codes for examples of *mdrip engine* implementing the first and the second target models can be found in Appendix A.

Chapter 5

Testing

The testing of the MDRIP development progressed from the initial testing, to the verification testing, and to the experimental testing.

Initially, the design focus was to implement an AKAROA2/PDNS linkage software. At a later verification stage, the first target model in Figure 3.8 with a queueing network of four independent queueing systems was arranged to verify the correctness of the native MDRIP implementation in AKAROA2. The native MDRIP implementation mainly refers to the *mdrip engine*, *subengine*, *AkSripObservation* routine, and other supporting routines. At the final experimental stage, the second target model with tandem queueing systems partitioned into distributed submodels was executed by the *mdrip engine* controlled by *akmaster* in AKAROA2.

Using the first target queueing network model with multiple independent queueing systems helps to verify the correctness of the MDRIP implementation without considering the correctness of modeling issues. The correctness of such MDRIP functionality refers to the correct preservation of message orderings for distributed messages.

It is important during the verification testing to separate the MDRIP

implementation issues and the modeling issues. Because such independency ensures that no sequencing constraint as yet no causality constraint existed among distributed *subengine*s. Therefore, values of message passings can be tested to confirm whether or not the MDRIP implementation is correctly preserved the message orderings of distributed *subengine*s.

However, the first target queueing network model in Figure 3.8 does not show how the MDRIP implementation in AKAROA2 handles more complicated modeling issues, such as potential sequencing and/or causality constraints. The second target queueing network model in Figure 3.9 with tandem connection provides a sequencing constraint to test whether or not the overall MDRIP implementation is still valid.

The experimental testing shows that the MDRIP implementation is valid provided the sequencing constraint introduced from the tandem connection model. The experimental results justify that the MDRIP implementation successfully creates a platform where distributed simulation can be arranged to run in AKAROA2 with on-line stochastic quantitative and sequential control on statistical errors. However, more complicated sequencing and/or causality constraints should be resolved by *subengine*s, not by the new MDRIP functionality.

5.1 Initial Testing

The initial design focus was motivated by the development of the Parallel/Distributed NS2 - PDNS [2] as well as the AKAROA2/NS2 linkage software [22]. The initial idea was to develop an AKAROA2/PDNS linkage software to verify the MDRIP approach. Initial testing mainly comprised installation and testing of both the PDNS and the AKAROA2/NS2 linkage software in the Linux environment at the CSSE. The purpose was to assess

compatibility and other supporting issues.

The PDNS reuses a huge database of networking modeling scripts available for the sequential NS [2, 21]. Not only are these networking modeling scripts freely available but also very popular and important in networking research. The AKAROA2/NS2 linkage software runs the sequential NS2 under the control of MRIP in AKAROA2. One example [23] using such modeling script from NS2 via the linkage software with the multiple replications of MRIP in AKAROA2 demonstrates the benefits of on-line sequential control on statistical accuracy from such interoperation.

PDNS uses a federated simulation approach and a blocking based conservative synchronisation [2]. To use PDNS, each time one needs to prepare a specific block of code in an `Otcl` script in order to specify the routing topology related to a specific distributed simulation model. This step is not very efficient and time consuming.

The webpage for the AKAROA2/NS2 linkage [22] reports some link errors using the new interface of `ns-2 v.2.26`. This interface covers the required features of parallelisation in PDNS. At the initial stage of this thesis work, it was found that by commenting out some blocks of code in the AKAROA2/NS2 linkage software, such link errors disappeared. However, such new features of parallelisation in PDNS can not be integrated directly MRIP in AKAROA2.

In further attempt to work around the PDNS software and to conduct an on-going survey on PDES, we learnt that the native implementation of MDRIP in AKAROA2 would be more feasible than the implementation of the AKAROA2/PDNS linkage, especially in consideration of reusing on-line sequential control functionality already implemented and well-tested in AKAROA2. Therefore, a design decision was made to shift the focus from

the AKAROA2/PDNS linkage to the native implementation of MDRIP in AKAROA2. Such implementation is mainly realised by the *mdrip engine*, *subengine*, and associated supporting routines.

The initial testing identifies:

- Priority of MDRIP development: Instead of linking AKAROA2 and PDNS, the sequential control features of MRIP should be extended first to a simple distributed simulation. Because on-line sequential control of statistical errors has been well implemented in AKAROA2, reusability will enhance implementation efficiency.

- Key features of MDRIP for design and implementation include conceptual overviews, target models, networking architecture, system components, as well as related processes and routines.

- How MDRIP is different from the other PDES approaches in that on-line output data analysis is sequentially controlled, so that statistical errors can be tested.

- Separation of communication topology and observation data flows features *publish and subscribe* connection management and well-defined message formats for message passings. It provides flexibility in communication topology and supports greater interoperability in the future.

5.2 Verification Testing

The first queueing network model with four independent queueing systems is arranged into two base cases to verify the correctness of the MDRIP implementation. Specifically, the sequencing orders of data observations sent from each *subengine* to associated *mdrip engine* need to be preserved when

mdrip engine receives and processes these distributed messages. Such order preservation is essential to the extension of quantitative and sequential control from MRIP to MDRIP, so that *mdrip engine* can process ordered data observations from distributed submodels.

The independence of such queueing systems is designed to exclude modeling issues during the verification of the correctness of the MDRIP implementation. Since data does not exchange in between queues, when the first queueing network model is partitioned into four submodels executed over four processors in parallel, there is no message passings required in between the four processors. The **compose** function which merges multiple incoming streams of data observations from each LP needs to reflect such semantic relationship. In the second base case, such **compose** function expects to receive a stream of data combinedly from four *subengine*s and directly relays this stream of data to *akmaster* via the `AkObservation` routine.

Each *subengine* is equivalent to an LP. Each incoming stream of data observations from a LP is received by *mdrip engine* and stored into a linked list queue. If the incoming stream of data has a sequence, then such linked list queue can preserve the data sequence. The goal of the verification testing is to see whether such data sequence is still preserved in partitioned and distributed case.

5.2.1 Base Cases

The four independent queueing systems are arranged as each a M/M/1 queue. As seen in Figure 3.8, there is only one source of customers randomly arrives in the network. When a customer arrives in the network, he/she is assigned with equal probability to one of the four queues to wait for service from the allocated queueing server. The service time of each queueing

server is exponentially distributed. The customer leaves the network when the service is finished.

In the non-partitioned and non-distributed case of Figure 5.1, there is only one LP running on one processor for one *subengine* running all sub-models. All streams of random numbers are allocated to this processor. And requests of random numbers from this *subengine* can only come from this processor. As a result, the *compose* function only handle one linked list queue in this base case.

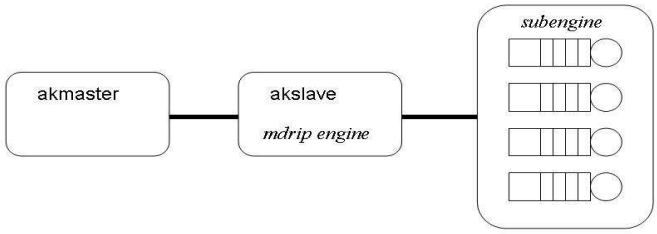

Figure 5.1: Non-partitioned and Non-distributed Base Case

If such queueing system is partitioned into subsystems, four queues in parallel are partitioned into four subsystems. Each subsystem runs on one processor. Figure 3.12 shows the model partitioning. Each subsystem represented by a LP behaves as one M/M/1 queue. Because they are networked in parallel and independent of each other, no data exchange required among these queues. Such semantics depict no specific message ordering of data observation required in between each parallel processors.

In other words, the order preservation only matters in each incoming distributed data stream from each LP. Because the incoming distributed data stream is observation data with timestamp values recording when the associated events been processed.

For the partitioned and distributed case of Figure 5.2, multiple streams of random numbers are allocated for each LP that runs a subsystem, since each LP requests random numbers via the *mdrip engine* by itself. Four linked list queues are needed for *mdrip engine* to `compose`.

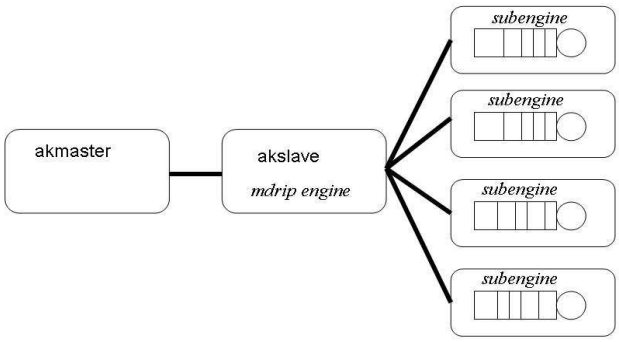

Figure 5.2: Partitioned and Distributed Base Case

The verification testing considers only the implementation correctness of the native MDRIP approach and isolates the modelling issues that could interfere such correctness. Because the four independent M/M/1 queues in parallel do not present the characteristic that subevent data distribute and exchange among each others. It could be added that the modelling scenario of the base cases are too artificial to be considered realistic. Nevertheless, this verification testing helps to set out the boundary of MDRIP features.

MDRIP processes correctly distributed data observation and assumes that observation data contains no causality errors. Because causality errors should have been resolved before such observation data is collected and sent to *mdrip engine*.

We argue that the design principles of the MDRIP implementation are applicable in the case of distributed simulation models with potential causality errors. The design principles of MDRIP are referred to as the supports of on-line stochastic quantitative and sequential control of simulation output data extended from *akmaster* of MRIP. MDRIP expects distributed observation data which is the simulation output data from distributed *subengine*s.

5.2.2 The Compositions

As each LP sends observed data via the *mdrip engine* to the *akmaster*, the *mdrip engine* receives the observed data and buffers the data into each linked list queue. The *mdrip engine* then dequeues the data from each linked list queue and merges these data according to specific semantic rules which reflect the model behaviours. Because the linked list queues preserve the message ordering when the subevents send observed data to the *mdrip engine*, the **compose** function in the *mdrip engine* needs to coordinate the correct set of disjoint subsets.

Figure 5.3 demonstrates what to compose in the case of four linked list queues.

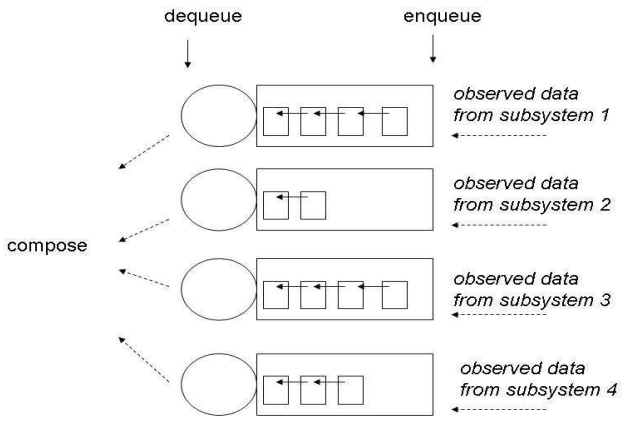

Figure 5.3: Compose Four Linked List Queues

Referring to Figure 3.12 and Figure 5.3 for the four independent M/M/1 queues in parallel, the **compose** feature is captured by following code snippet in the `Srip()` loop in *mdrip engine*:

```
if (lr1 != NULL && lr2 != NULL && lr3 != NULL && lr4 != NULL) {
    v1 = lr1->data1; v2 = lr2->data1; v3 = lr3->data1; v4 = lr4->data1;
    t1 = lr1->data2; t2 = lr2->data2; t3 = lr3->data2; t4 = lr4->data2;
    if (t1 <= t2 && t3 <= t4) {
        if (t1 <= t3)
            lr1 = remove(lr1); AkObservation(v1);
        else
            lr3 = remove(lr3); AkObservation(v3);
        }
    if (t1 > t2 && t3 <= t4) {
        if (t2 <= t3)
            lr2 = remove(lr2); AkObservation(v2);
        else
            lr3 = remove(lr3); AkObservation(v3);
```

```
        }
    if (t1 <= t2 && t3 > t4) {
        if (t1 <= t4)
            lr1 = remove(lr1); AkObservation(v1);
        else
            lr4 = remove(lr4); AkObservation(v4);
        }
    if (t1 > t2 && t3 > t4) {
        if (t2 <= t4)
            lr2 = remove(lr2); AkObservation(v2);
        else
            lr4 = remove(lr4); AkObservation(v4);
        }
}
```

Four linked list queues are arranged for the *mdrip engine*: `lr1, lr2, lr3,` and `lr4`. `v1, v2, v3, v4` are the observation values dequeued from the linked list queues in *mdrip engine* enqueued by observation data from incoming subsystems. `t1, t2, t3, t4` are the distributed timestamp values dequeued from the linked list queues in *mdrip engine* enqueued by observation data from incoming subsystems. These distributed timestamp values are given by each *subengine* and are actually representing the ordering sequency required to be preserved. Therefore, the four inside `if` statements are implemented to compare and arrange the correct ordering sequency of the incoming data elements before dequeuing them and passing them to `AkObservation`. Because the four subsystems are independent and no direct data flow among each others, the **compose** feature is simply to relay the values of dequeued data observations and directly pass into the `AkObservation` routine. No extra operation is required here.

In general, the `Srip()` loop in *mdrip engine* serves as one common input

of random numbers, because each *subengine* required to send request through and receive allocated random numbers from the *mdrip engine*. Above outer `if` statement can only be executed when all the linked list queues are not empty. It means that on-line sequential control of statistical errors is executed in parallel with the *subengine*s producing data observations. The event list management is handled by the *subengine*s which include `process.H` and `resource.H` rather than by the *mdrip engine*. In other words, *mdrip engine* does not handle the event list management, because *mdrip engine* does not run the simulation model.

5.2.3 General Verification

From previous code snippet, it is observed that such `compose` feature needs to merge sets of dequeued data from multiple linked list queues. Such **compose** function is the key feature of the MDRIP implementation. As the number of the linked list queues increases, the efficiency of such merge implementation will be critical.

Following three questions are asked to test such verification. Previous two questions concern whether the *mdrip engine* receives correct data observation from each *subengine*. The third question concerns whether the **compose** function itself is working correctly in *mdrip engine*.

- *Are the data formats of MDRIP messages correct?*

  If the formats of M_RNDQ and M_RNDA are correctly reused as well as the format of M_OBSV is well implemented under existing MRIP function, then the data formats of MDRIP messages are correct. If the data format of a message is not correct, then message passings are based on meaningless data.

- *Does the AkSripObservation routine correctly intercommunicate MDRIP*

messages between the mdrip engine and subengines?

`AkSripObservation` sends observed data from each *subengine* to the *mdrip engine*. Adding a *counter* function in the `AkSripObservation` routine which numbers each sending message can test whether the message ordering is preserved after the observed data is sent to the *mdrip engine*.

- *Does the mdrip engine correctly coordinate MDRIP messages?*

  Such coordination is done by the **compose** feature implemented in the *mdrip engine* after data observation is received and buffered into each linked list queue. How to compose these dequeued data defines the **compose** features that should reflect the semantic logic of simulated model.

The correctness of the MDRIP implementation for the base cases can be tested via the values of the *counter* function incremented at each *subengine* and sent via the third parameter of `AkSripObservation` to be received by *mdrip engine*. For the case of the four independent queues, the sequence of `1,1,1,1,2,2,2,2,3,3,3,3,...` is collected. The values of the sequence of numbers confirm the order-preserving feature of using linked list queues for incoming messages distributedly from each *subengine*.

The following experimental testing attempts to add modelling issues. For the second target model of a queueing network with two queueing systems connected in tandem, a socket connection is established between two processors, each running a simulation of M/M/1 queue, by a *subengine*. Data is passed through such socket connection from one processor to the other and data observation is collected distributedly from each *subengine*. Thus, the logic of the **compose** feature needs to be modified. As a result, impacts from the modelling issues can be examined under the MDRIP approach.

5.3 Experimental Testing

5.3.1 Testing Scenarios

The second target model of a queueing network with two M/M/1 queues in tandem is executed under various parallelisation approaches. From the perspective of testing design, several scenarios are described in diagrams. A round-edged rectangle represents one single processor. A line between two round-edged rectangles represents a communication channel, such as a socket connection. N denotes the number of replications.

The purpose of the experimental testing is to evaluate whether results of estimates referring to three statistics of interests are valid under different testing scenarios. The three statistics of interests discussed in Chapter 3.2 are total mean waiting time, total mean service time, and total mean response time.

Based on the queueing network of two M/M/1 queues connected in tandem, the theoretical values are calculated for system utilisation ρ ranging from 0.1 up to 0.9. Raw figures in details are provided in Appendix B. The final experimental results of on-line simulation show that the MDRIP implementation is applicable in the case of distributed simulation. As a result of on-line stochastic quantitative and sequential control on automated output data analysis, graphs related to parameters of interests are plotted and discussed.

Brief discussions concerning speedup are also included, however, evaluation of speedup is suggested for the future work.

1. MRIP base case, $N = 1$

   Figure 5.4 demonstrates the scenario where the MRIP approach runs the simulation model on only one processor, therefore, only one *akslave*, one *engine*, and one replication. The tandem queueing network is not distributed, because the whole model is running on one processor. This MRIP base case is usually used for testing of AKAROA2 implementation.

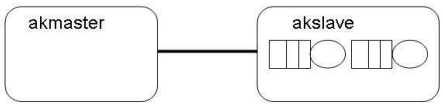

Figure 5.4: MRIP, $N = 1$

Since only one processor is utilised, the speed/time measured in this case serves as a base unit to compare with other potential speedup. This base case itself is just an on-line stochastic DES.

2. MRIP, $N > 1$

   The MRIP approach runs the simulation model over multiple processors as shown in Figure 5.5. More than one *akslave*s and *engine*s produce multiple replications. The target model run in each replication is still not distributed simulation, because the whole model is running on each processor. As $N > 1$, multiple replications are where statistical

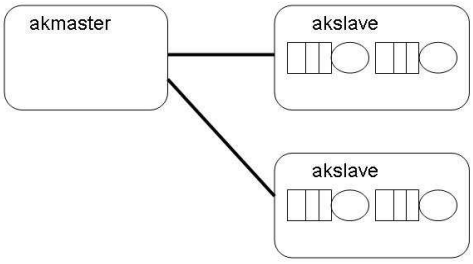

Figure 5.5: MRIP, $N > 1$

speedup can be achieved. The speed/time measured in this $N > 1$ case is used to compare with the MRIP base case $N = 1$ to evaluate whether the statistical speedup can be obtained if N is increased [4, 5].

3. MDRIP non-distributed base case

   The MDRIP non-distributed base case shown in Figure 5.6 is mainly for the testing of MDRIP implementation. Because there is only one *subengine* running the whole tandem queueing model, however, *mdrip engine* is used to receive observation data from the only one *subengine* and send such data observation via `AkObservation` to *akmaster*.

Figure 5.6: Non-distributed MDRIP

The comparison between this MDRIP non-distributed base case and the MRIP base case should show the basic computation overhead differs between MRIP and MDRIP. The speed/time measured in the MDRIP non-distributed base case is expected to be more than the speed/time measured in the MRIP base case. Such information should reflect the cost of introducing the *mdrip engine*.

4. MDRIP base case with distributed simulation

   The MDRIP base case is shown in Figure 5.7 where the single replication is performed by only one *akslave* launching only one *mdrip engine*. The simulation model is partitioned and distributed in this case. Because there are more than one *subengine*s. Each queueing system in the tandem queueing network is executed on one particular *subengine*. As a subevent of one customer leaving the first queue simulated on the first *subengine*, the subevent of this customer that arrives at the second queue is distributed to the second *subengine* and simulated on the second *subengine*.

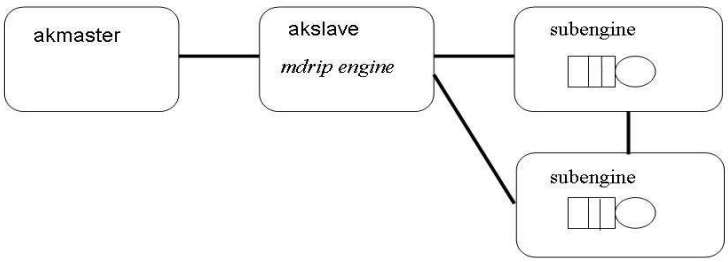

Figure 5.7: Distributed MDRIP, $N = 1$

In this case, *mdrip engine* processes observed data distributedly from distributed simulation executed on two parallel processors.

To compare the speed/time between the MDRIP distributed base case and the MDRIP non-distributed base case should distinguish the com-

putational overheads of the same simulation model but in non-distributed and distributed cases. This comparison excludes the overhead factor of *mdrip engine*.

To compare the speed/time between the MDRIP distributed base case and the MRIP base case should distinguish the computation overheads of the same simulation model but in non-distributed and distributed cases. This comparison includes the overhead factor of *mdrip engine*.

5.3.2 Specific Verification

As modelling issues are considered, following two questions are raised:

- *Does the subengine correctly apply existing modeling library of AKAROA2 and use AkSripObservation routine to report observation results to the mdrip engine?*

  If the parameters of interests are compared between the theoretical results and the experimental results, they should exhibit very close similarity. The experimental results at the following subsection prove this to be true. Theoretical results are close to the experimental results, therefore, the MDRIP implementation does correctly support existing modeling library of AKAROA2.

- *Are results of MDRIP simulation consistent with expectation of MDRIP requirements and designs?*

  The parameters of interests for this experimental testing include the total mean waiting time, the total mean service time, and the total mean response time in the queueing network system with two M/M/1 queueing systems connected in tandem refering to the second target model.

$TotalMeanWaitingTime + TotalMeanServiceTime$

$= TotalMeanResponseTime$

It can be observed that the values from the total mean waiting time plus the values from the total mean service time are about the same as the total mean response time.

After theoretical results calculated and experimental results are collected, these results of output data analysis in the next subsection show that MDRIP development in this thesis work meets the expectation of MDRIP requirements and designs.

5.3.3 Experimental Results

In all the cases, Total Mean Waiting Time (Figure 5.8), Total Mean Service Time (Figure 5.9), and Total Mean Response Time (Figure 5.10), MRIP N=1 is slightly different from MRIP N=2, because different streams of random number are requested and allocated. MRIP N=1 is the same as Non-distributed MDRIP because they are actually using the same set of random numbers. Non-distributed MDRIP has *mdrip engine* in the *server&client* side to relay such random number messages. Distributed MDRIP is also slightly different from the other set of data because the random numbers requested and allocated to the first *subengine* is different from the random numbers dispatched to the second *subengine*. In addition, the waiting time T_{2W} in S2 of Figure 3.9 is subject to when a customer leaves at t_{1d}. Such dynamic behaviour is reflected by the distributed *subengine*s.

All the experiments are performed with the service rate at 10 for each queue, or the mean service time equals to 0.1 at each queue. Therefore, two queues connected in tandem are expected the service rate to be 20, or equivalently the mean service time at 0.2.

1. Total Mean Waiting Time

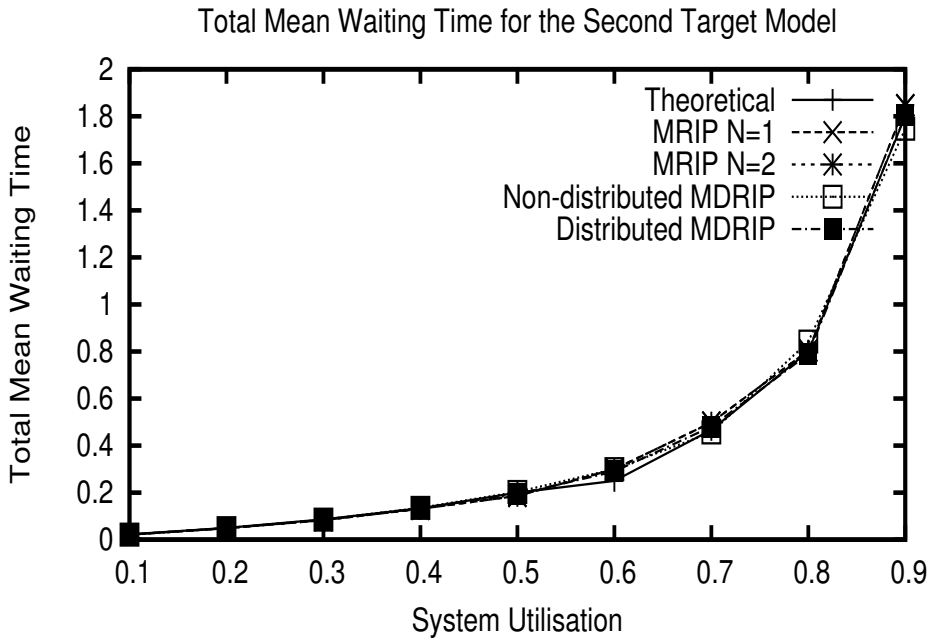

Figure 5.8: Total Mean Waiting Time for the Second Target Model

Figure 5.8 shows that five sets of total mean waiting time appear to be the results of correct simulations. As the system gets more heavily utilised, the total mean waiting time is increased exponentially. Similar to one single M/M/1 queueing system, whereas, the amount of time is double. It corresponds to two M/M/1 queueing systems connected in tandem which reflects the addition composition discussed before related to the second target model.

2. Total Mean Service Time

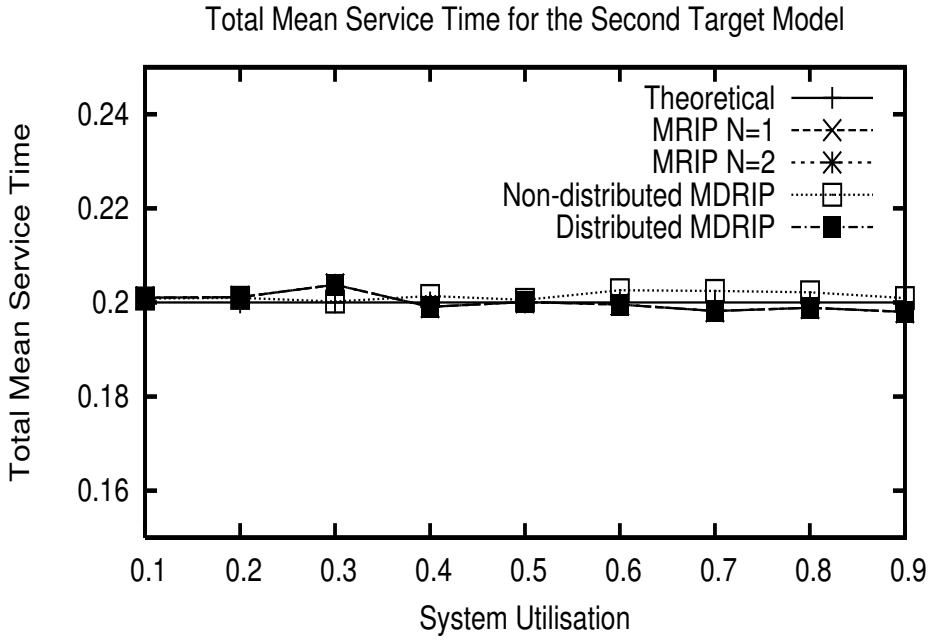

Figure 5.9: Total Mean Service Time for the Second Target Model

In the case of Total Mean Service Time in Figure 5.9, all five sets of data are close to each other because the total service time combined from the first *subengine* and the second *subengine* are independent from each other. These service time values are dependent on the distribution of service rate. Therefore, as the system becomes busy, the total service time will not change much. Theoretical values are constant as the system utilisation changed. The total mean service time are correct simulation results which are nearly twice of the mean service time 0.1 at each queue indicating the addition composition.

3. Total Mean Response Time

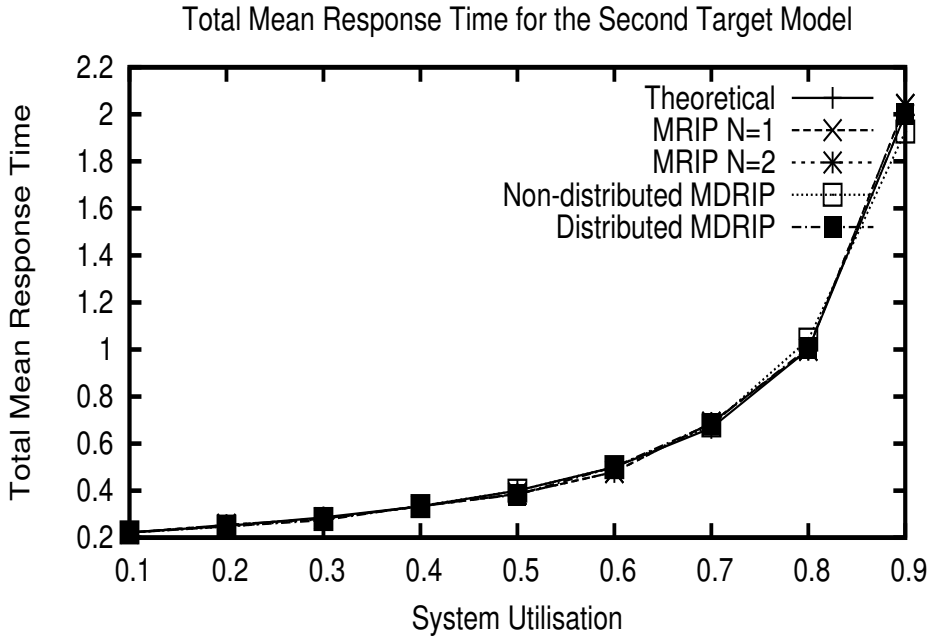

Figure 5.10: Total Mean Response Time for the Second Target Model

Figure 5.10 shows that five sets of total mean response time are correct simulation results following the approximate sum of the total mean waiting time and the total mean service time. As expected, the total mean response time is increased exponentially as the system utilisation ρ is varied from 0.1 to 0.9 and the values of the total mean response time are nearly twice of the mean response time for a single M/M/1 queue.

Raw data can be referred to Appendix B.

5.4 Summary

This chapter discusses testing issues of MDRIP development cover initial testing, verification testing, and experimental testing.

The initial testing found out that implementation of native *mdrip engine* is more feasible than implementation of AKAROA2/PDNS linkage, especially in the reuse of on-line sequential control functionality. The implementation of network topology and data flows is better kept separate.

The verification testing analyses the correct preservation of ordering sequency of distributed data elements among multiple linked list queues in *mdrip engine*. The composition of the first target model is verified. The result showed the design and implementation of *mdrip engine*, *subengine*s, and other supporting routines are correct.

The experimental testing tested the second target model based on four different testing scenarios. Three statistical estimates are discussed: The Mean Waiting Time, the Total Mean Service Time, and the Total Mean Response Time. Theoretical values are calculated to compare experimental results with four different testing scenarios. It shows that MDRIP is possible to support on-line output data analysis with sequential control of statistical errors when the simulation is partitioned and distributed.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The MDRIP approach consists of the design and implementation of an *mdrip engine* that extends quantitative and sequential control services from MRIP controlled by *akmaster*. Two-stage I/O multiplexing provides the effective framework of interprocess communication required by MDRIP. Random numbers messages and data observation messages are successfully distributed among *subengines* via *mdrip engine*.

The development of MDRIP is a series of efforts attempting to interoperate sequential on-line simulation with automated statistical inference and distributed simulation with partitioned submodels. MDRIP is able to support quantitative and sequential analysis with flexible run lengths as well as analyse different parameters from different submodels at a controllable manner. It is important that observation data is uncorrelated whether the simulation is non-distributed or distributed. Simulation credibility is guaranteed by adaptively adjusting the level of statistical errors. Large-sized and complex simulation models can be divided into smaller submodels for

detailed examination and the interaction between submodels assists in understanding dynamic system behavior.

The main contribution of this thesis can be considered as establishing a base platform for exploring MDRIP simulation in AKAROA2.

Modeling issues are isolated in verification testing based on the queueing network of multiple independent queueing systems. Experimental testing includes modeling issues based on validated MDRIP platform. The queueing network with two M/M/1 queues connected in tandem is used to experiment the implementation of MDRIP. Mean values of the estimates: total waiting time, total service time, and total response time are produced in four different parallelisation conditions as well as in comparison of theoretical calculation. The results demonstrate that the design and implementation of MDRIP works effectively and produces correct simulation results.

In conclusion, the *mdrip engine* successfully provides the base for exchanging necessary messages passings of on-line sequential simulation analysis extended from MRIP to MDRIP. *AkSripObservation* is the new interface routine for submodels to submit distributed data observations for the stochastic on-line simulation managed by *akmaster*.

6.2 Future Work

During the development of MDRIP, following ideas have been identified for possible future work.

- In extension from the base platform, testing MDRIP with multiple *mdrip engine*s.
- Possible interoperation between *mdrip engine* and HLA/RTI contracts.
- Evaluation of larger and more complex simulation models.

Appendix A

Source Code

1. mysim.C *mdrip engine* for the first target model

```
#include <akaroa.H>
#include <akaroa/distributions.H>
#include <akaroa/exit.H>
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
#include <netdb.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/param.h>
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
#include "gethostname.H"
#include "ipc/connection.H"
#include "message.H"
#include "args.H"
#include "slave_to_engine.H"
#include "master_to_client.H"
#include "ipc/error.H"
#include "debug.H"
#include "akaroa/exit.H"
#include <sys/socket.h>
```

```
#include <sys/select.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <time.h>
#include "simulation.H"
#include "boolean.H"
#include "environment.H"
#include "tagged_block.H"
#include "sripslavemaster_to_client.H"
#include "client_to_sripslavemaster.H"
#include "checkpoint.H"
#include "akaroa.H"
#include "../engine/engine_to_master.H"
#include "client_to_master.H"
#include "akaroa/process.H"
#include "AkSripObservation.H"
#include "../engine/engine_environment.H"
extern "C" { int getdtablesize(); }
int srip_max_fds; /* Max file descriptor + 1 */
Connection      **srip_connections;  /* Mapping from socket fd -> handler */
sockaddr_in srip_master_addr; /* Address bound to listen socket */
int srip_listen_sock; /* Socket for accepting connections */
fd_set srip_select_rd_fds; /* File descriptors to select for reading */
fd_set srip_select_wr_fds; /* File descriptors to select for writing */
Connection *m = 0;
void InitSripTables();
void InitSripSockets();
void InitSripMasterAddress();
void Srip();
int main(int argc, char *argv[]) {
  InitSripTables();
  InitSripSockets();
  InitSripMasterAddress();
  m = GetMasterConnection();
  Srip();
}
void InitSripTables()
```

```
{
  srip_max_fds = FD_SETSIZE;
  srip_connections = new Connection*[srip_max_fds];
  for (int i=0; i<srip_max_fds; i++)
    srip_connections[i] = 0;
}
void InitSripSockets()
{
  socklen_t srip_namelen = sizeof(srip_master_addr);
  srip_listen_sock = socket(AF_INET, SOCK_STREAM, 0);
  if (srip_listen_sock < 0)
    goto bad;
  srip_master_addr.sin_family = AF_INET;
  srip_master_addr.sin_addr.s_addr = INADDR_ANY;
  srip_master_addr.sin_port = 0;
  if (bind(srip_listen_sock, (sockaddr *)&srip_master_addr, sizeof(srip_master_addr)))
    goto bad;
  if (getsockname(srip_listen_sock, (sockaddr *)&srip_master_addr, &srip_namelen) < 0)
    goto bad;
  if (listen(srip_listen_sock, 100) < 0)
    goto bad;
  FD_ZERO(&srip_select_rd_fds);
  FD_SET(srip_listen_sock, &srip_select_rd_fds);
  return;
 bad:
  perror("sripslave: Failed to create srip listen socket");
  Exit(1);
}
static void KeepSripSocketAlive(int fd)
{
  int value = 1;
  setsockopt(fd, SOL_SOCKET, SO_KEEPALIVE, (char *)&value, sizeof(value));
}
struct llq{
  double data1, data2;
  struct llq *next;
};
typedef struct llq qlist;
qlist * add(qlist *lptr, double d1, double d2);
```

```
qlist * remove(qlist * lptr);
void clearqueue(qlist * lptr);
qlist * add(qlist *lptr, double d1, double d2) {
  qlist * lp = lptr;
  if (lptr != NULL) {
    while (lptr -> next != NULL)
      lptr = lptr -> next;
    lptr -> next = (qlist *) malloc (sizeof (qlist));
    lptr = lptr -> next;
    lptr -> next = NULL;
    lptr -> data1 = d1;
    lptr -> data2 = d2;
    return lp;
  } else {
    lptr = (qlist *) malloc (sizeof (qlist));
    lptr -> next = NULL;
    lptr -> data1 = d1;
    lptr -> data2 = d2;
    return lptr;
  }
}
qlist * remove(qlist * lptr) {
  qlist * tp;
  tp = lptr -> next;
  free (lptr);
  return tp;
}
qlist *lr1 = NULL;
qlist *lr2 = NULL;
qlist *lr3 = NULL;
qlist *lr4 = NULL;
void Srip() {
  int srip_fd;
  int s_fd[2];
  for (;;) {
    fd_set read_fds = srip_select_rd_fds;
    fd_set write_fds = srip_select_wr_fds;
    int result = select(srip_max_fds+1, &read_fds, &write_fds, NULL, NULL);
    if (result < 0) {
```

```
      perror("mysim: select");
      //exit(1); }
    if (result == 0) {}
    if (result > 0) {
      for (srip_fd = 0; srip_fd < srip_max_fds+1; srip_fd++) {
if (FD_ISSET(srip_fd, &read_fds)) {
  if (srip_fd == srip_listen_sock) {
    sockaddr srip_addr;
    socklen_t srip_addrlen = sizeof(srip_addr);
    int d_fd;
    Connection *srip_c = 0;
    d_fd = accept(srip_listen_sock, &srip_addr, &srip_addrlen);
    KeepSripSocketAlive(d_fd);
    srip_c = new Connection(d_fd);
    FD_SET(d_fd, &srip_select_rd_fds);
    srip_connections[d_fd] = srip_c;
    } else if (srip_fd == m->fd) {
    if (debug) {
fprintf(debug_file, "SripRecvingMasterConnection: Recv: ");
ReportError(debug_file); }
    //goto bad;
  } else {
    Message srip_msg;
    char srip_buf[MAX_MSG_LEN];
    if (srip_connections[srip_fd]->Recv(srip_msg, srip_buf, sizeof(srip_buf)) < 0)
      {
if (debug) {
    fprintf(debug_file, "AcceptSripConnection: Recv: ");
    ReportError(debug_file); }
FD_CLR(srip_fd, &srip_select_rd_fds);
  delete srip_connections[srip_fd];
srip_connections[srip_fd] = NULL;
      } else {
      if (srip_msg == M_VREQ) {
Environment *env = EngineEnvironment();
env->SendTo(srip_connections[srip_fd]);
      } else {
switch (srip_msg)
  {
```

```
  case M_NPAR:
    break;
  case M_RNDQ:
    m->Send(M_RNDQ);
    Message msg_s;
    char buf_s[MAX_MSG_LEN];
    int exp;
    unsigned long mant;
    if (m->Recv(msg_s, buf_s, sizeof(buf_s)) < 0) {
perror("m->Recv(msg_s, buf_s, sizeof(buf_s)");
//goto bad;
    } else {
char * buffer;
buffer = getenv("HOST");
if (msg_s == M_RNDA) {
    sscanf(buf_s, "%d %lu", &exp, &mant);
    srip_connections[srip_fd]->Send(msg_s, buf_s); }
    }
    break;
  case M_OBSV:
    int n;
    double x, t;
    double v1, v2, v3, v4, t1, t2, t3, t4;
    sscanf(srip_buf, "%d %lg %lg", &n, &x, &t);
    if (srip_fd == 7)
      lr1 = add(lr1, x, t);
      if (srip_fd == 9)
      lr2 = add(lr2, x, t);
    if (srip_fd == 10)
      lr3 = add(lr3, x, t);
    if (srip_fd == 11)
      lr4 = add(lr4, x, t);
    if (lr1 != NULL && lr2 != NULL && lr3 != NULL && lr4 != NULL) {
      v1 = lr1->data1;
      v2 = lr2->data1;
      v3 = lr3->data1;
      v4 = lr4->data1;
      t1 = lr1->data2;
      t2 = lr2->data2;
```

```
      t3 = lr3->data2;
      t4 = lr4->data2;
      if (t1 <= t2 && t3 <= t4) {
if (t1 <= t3) {
  lr1 = remove(lr1);
  AkObservation(v1);
                        } else {
  lr3 = remove(lr3);
  AkObservation(v3); }
                      }
      if (t1 > t2 && t3 <= t4) {
if (t2 <= t3) {
  lr2 = remove(lr2);
  AkObservation(v2);
} else {
  lr3 = remove(lr3);
  AkObservation(v3); }
      }
      if (t1 <= t2 && t3 > t4) {
if (t1 <= t4) {
  lr1 = remove(lr1);
  AkObservation(v1);
} else {
  lr4 = remove(lr4);
  AkObservation(v4); }
      }
      if (t1 > t2 && t3 > t4) {
if (t2 <= t4) {
  lr2 = remove(lr2);
  AkObservation(v2);
} else {
  lr4 = remove(lr4);
  AkObservation(v4); }
       }
    }
    break;
  default:
    error = E_BABL; //
    if (debug) {
```

```
fprintf(debug_file, "AcceptSripConnection: ");
ReportError(debug_file);
    }
}}}}}}}}
void InitSripMasterAddress() {
  char buf[MAX_MSG_LEN];
  char srip_host[32];
  int srip_port, srip_pid;
  if (GetSripslaveMasterAddress(srip_host, srip_port, srip_pid) == 0) {
      fprintf(stderr, "%s\n%s %d %s %s%s\n%s\n%s\n",
 "sripslave: There already seems to be an sripslave running as",
 "process", srip_pid, "on", srip_host, ". If that process no longer exists,",
 "remove the file '.sripslave' from your home directory and run",
 "sripslave again.");
      system("rm ~/.sripslave");
      //Exit(1); }
  if (SetSripslaveMasterAddress(ntohs(srip_master_addr.sin_port)) < 0) {
      if (error == E_EXIS) {
  ReportError("sripslave");
    //Exit(1); }}
}
```

2. **mysim.C** *mdrip engine* for the second target model

```
#include <akaroa.H>
#include <akaroa/distributions.H>
#include <akaroa/exit.H>
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
#include <netdb.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/param.h>
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
#include "gethostname.H"
#include "ipc/connection.H"
```

```
#include "message.H"
#include "args.H"
#include "slave_to_engine.H"
#include "master_to_client.H"
#include "ipc/error.H"
#include "debug.H"
#include "akaroa/exit.H"
#include <sys/socket.h>
#include <sys/select.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <time.h>
#include "simulation.H"
#include "boolean.H"
#include "environment.H"
#include "tagged_block.H"
#include "sripslavemaster_to_client.H"
#include "client_to_sripslavemaster.H"
#include "checkpoint.H"
#include "akaroa.H"
#include "../engine/engine_to_master.H"
#include "client_to_master.H"
#include "akaroa/process.H"
#include "AkSripObservation.H"
#include "../engine/engine_environment.H"
extern "C" { int getdtablesize(); }
int srip_max_fds; /* Max file descriptor + 1 */
Connection      **srip_connections;  /* Mapping from socket fd -> handler */
sockaddr_in srip_master_addr; /* Address bound to listen socket */
int srip_listen_sock; /* Socket for accepting connections */
fd_set srip_select_rd_fds; /* File descriptors to select for reading */
fd_set srip_select_wr_fds; /* File descriptors to select for writing */
Connection *m = 0;
void InitSripTables();
void InitSripSockets();
void InitSripMasterAddress();
void Srip();
```

```
int main(int argc, char *argv[]) {
  file_x = fopen("X.txt","w");
  file_y1 = fopen("Y1.txt","w");
  file_y2 = fopen("Y2.txt","w");
  InitSripTables();
  InitSripSockets();
  InitSripMasterAddress();
  m = GetMasterConnection();
  Srip();
  fclose(file_x);
  fclose(file_y1);
  fclose(file_y2);
}
void InitSripTables()
{
  srip_max_fds = FD_SETSIZE;
  srip_connections = new Connection*[srip_max_fds];
  for (int i=0; i<srip_max_fds; i++)
    srip_connections[i] = 0;
}
void InitSripSockets()
{
  socklen_t srip_namelen = sizeof(srip_master_addr);
  srip_listen_sock = socket(AF_INET, SOCK_STREAM, 0);
  if (srip_listen_sock < 0)
    goto bad;
  srip_master_addr.sin_family = AF_INET;
  srip_master_addr.sin_addr.s_addr = INADDR_ANY;
  srip_master_addr.sin_port = 0;
  if (bind(srip_listen_sock, (sockaddr *)&srip_master_addr, sizeof(srip_master_addr)))
    goto bad;
  if (getsockname(srip_listen_sock, (sockaddr *)&srip_master_addr, &srip_namelen) < 0)
    goto bad;
  if (listen(srip_listen_sock, 100) < 0)
    goto bad;
  FD_ZERO(&srip_select_rd_fds);
  FD_SET(srip_listen_sock, &srip_select_rd_fds);
  return;
 bad:
```

```
  perror("sripslave: Failed to create srip listen socket");
  Exit(1);
}
static void KeepSripSocketAlive(int fd)
{
  int value = 1;
  setsockopt(fd, SOL_SOCKET, SO_KEEPALIVE, (char *)&value, sizeof(value));
}
struct llq{
  double data1, data2;
  struct llq *next;
};
typedef struct llq qlist;
qlist * add(qlist *lptr, double d1, double d2);
qlist * remove(qlist * lptr);
void clearqueue(qlist * lptr);
qlist * add(qlist *lptr, double d1, double d2) {
  qlist * lp = lptr;
  if (lptr != NULL) {
    while (lptr -> next != NULL)
      lptr = lptr -> next;
    lptr -> next = (qlist *) malloc (sizeof (qlist));
    lptr = lptr -> next;
    lptr -> next = NULL;
    lptr -> data1 = d1;
    lptr -> data2 = d2;
    return lp;
  } else {
    lptr = (qlist *) malloc (sizeof (qlist));
    lptr -> next = NULL;
    lptr -> data1 = d1;
    lptr -> data2 = d2;
    return lptr; }
}
qlist * remove(qlist * lptr) {
  qlist * tp;
  tp = lptr -> next;
  free (lptr);
  return tp;
```

```
}
qlist *lr1 = NULL;
qlist *lr2 = NULL;
void Srip() {
  int srip_fd;
  char * buffer;
  buffer = getenv("HOST");
  for (;;) {
    fd_set read_fds = srip_select_rd_fds;
    fd_set write_fds = srip_select_wr_fds;
    int result = select(srip_max_fds+1, &read_fds, &write_fds, NULL, NULL);
    if (result < 0) {
      perror("mysim: select");
      //exit(1); }
    if (result == 0) {}
    if (result > 0) {
      for (srip_fd = 0; srip_fd < srip_max_fds+1; srip_fd++) {
if (FD_ISSET(srip_fd, &read_fds)) {
  if (srip_fd == srip_listen_sock) {
    sockaddr srip_addr;
    socklen_t srip_addrlen = sizeof(srip_addr);
    int d_fd;
    Connection *srip_c = 0;
    d_fd = accept(srip_listen_sock, &srip_addr, &srip_addrlen);
    KeepSripSocketAlive(d_fd);
    srip_c = new Connection(d_fd);
    FD_SET(d_fd, &srip_select_rd_fds);
    srip_connections[d_fd] = srip_c;
    } else if (srip_fd == m->fd) {
    if (debug) {
fprintf(debug_file, "SripRecvingMasterConnection: Recv: ");
ReportError(debug_file); }
    //goto bad;
  } else {
    Message srip_msg;
    char srip_buf[MAX_MSG_LEN];
    if (srip_connections[srip_fd]->Recv(srip_msg, srip_buf, sizeof(srip_buf)) < 0) {
if (debug) {
    fprintf(debug_file, "AcceptSripConnection: Recv: ");
```

```
    ReportError(debug_file); }
FD_CLR(srip_fd, &srip_select_rd_fds);
  delete srip_connections[srip_fd];
srip_connections[srip_fd] = NULL;
      } else {
      if (srip_msg == M_VREQ) {
Environment *env = EngineEnvironment();
env->SendTo(srip_connections[srip_fd]);
      } else {
switch (srip_msg) // srip
  {
  case M_NPAR:
    break;
  case M_RNDQ:
    m->Send(M_RNDQ);
    Message msg_s;
    char buf_s[MAX_MSG_LEN];
    int exp;
    unsigned long mant;
    if (m->Recv(msg_s, buf_s, sizeof(buf_s)) < 0) {
perror("m->Recv(msg_s, buf_s, sizeof(buf_s)");
//goto bad;
    } else {
char * buffer;
buffer = getenv("HOST");
if (msg_s == M_RNDA) {
    sscanf(buf_s, "%d %lu", &exp, &mant);
    srip_connections[srip_fd]->Send(msg_s, buf_s); }}
    break;
  case M_OBSV:
    int n;
    double x, t;
    double v1, t1, v2, t2, v;
    sscanf(srip_buf, "%d %lg %lg", &n, &x, &t);
    if (srip_fd == 8)
      lr1 = add(lr1, x, t);
    if (srip_fd == 10)
      lr2 = add(lr2, x, t);
    if (lr1 != NULL && lr2 != NULL) {
```

```
      v1 = lr1->data1;
      t1 = lr1->data2;
      v2 = lr2->data1;
      t2 = lr2->data2;
      v = v1 + v2;
      lr1 = remove(lr1);
      lr2 = remove(lr2);
      AkObservation(v); }
    break;
  default:
    error = E_BABL; //
    if (debug) {
fprintf(debug_file, "AcceptSripConnection: ");
ReportError(debug_file); }
  }}}}}}}}
}
void InitSripMasterAddress() {
  char buf[MAX_MSG_LEN];
  char srip_host[32];
  int srip_port, srip_pid;
  if (GetSripslaveMasterAddress(srip_host, srip_port, srip_pid) == 0) {
      fprintf(stderr, "%s\n%s %d %s %s%s\n%s\n%s\n",
 "sripslave: There already seems to be an sripslave running as",
 "process", srip_pid, "on", srip_host, ". If that process no longer exists,",
 "remove the file '.sripslave' from your home directory and run",
 "sripslave again.");
      system("rm ~/.sripslave");
      system("rm ~/.sripslave1");
      //Exit(1); }
  if (SetSripslaveMasterAddress(ntohs(srip_master_addr.sin_port)) < 0) {
      if (error == E_EXIS) {
  ReportError("sripslave");
         //Exit(1); }}
}
```

Appendix B

Raw Data

Second Target Model

Total Mean Waiting Time, Confidence = 0.95, Precision = 0.05

System Utilisation (ρ)	Theoretical	MRIP N=1	MRIP N=2	Non-distributed MDRIP	Distributed MDRIP
0.1	0.0222	0.021741	0.0220228	0.021741	0.0220265
0.2	0.05	0.0490309	0.0486354	0.0490309	0.0489828
0.3	0.0858	0.0832042	0.0849586	0.0832042	0.0829591
0.4	0.1334	0.131632	0.135164	0.131632	0.134503
0.5	0.2	0.185845	0.202744	0.185845	0.193577
0.6	0.25	0.300044	0.299653	0.300044	0.290347
0.7	0.4666	0.497983	0.456182	0.497983	0.480533
0.8	0.8	0.80436	0.841997	0.80436	0.788964
0.9	1.8	1.84751	1.74727	1.84751	1.80531

Total Mean Service Time, Confidence = 0.95, Precision = 0.05

System Utilisation (ρ)	Theoretical	MRIP N=1	MRIP N=2	Non-distributed MDRIP	Distributed MDRIP
0.1	0.2	0.201042	0.20085	0.201042	0.201042
0.2	0.2	0.201136	0.200965	0.201136	0.201136
0.3	0.2	0.203757	0.200201	0.203757	0.203757
0.4	0.2	0.199059	0.201397	0.199059	0.199059
0.5	0.2	0.20012	0.200558	0.20012	0.20012
0.6	0.2	0.199555	0.202608	0.199555	0.199555
0.7	0.2	0.198174	0.20245	0.198174	0.198174
0.8	0.2	0.198875	0.202162	0.198875	0.198875
0.9	0.2	0.19801	0.200941	0.19801	0.19801

Total Mean Response Time, Confidence = 0.95, Percision = 0.05

System Utilisation (ρ)	Theoretical	MRIP N=1	MRIP N=2	Non-distributed MDRIP	Distributed MDRIP
0.1	0.2222	0.221834	0.22281	0.221834	0.221834
0.2	0.25	0.254168	0.247477	0.254168	0.24824
0.3	0.2858	0.281816	0.279767	0.281816	0.27435
0.4	0.3334	0.333758	0.334333	0.333758	0.336464
0.5	0.4	0.386135	0.400286	0.386135	0.383055
0.6	0.5	0.479794	0.50004	0.479794	0.500497
0.7	0.6666	0.688089	0.677462	0.688089	0.684553
0.8	1	0.99884	1.04254	0.99884	1.00735
0.9	2	2.03915	1.9264	2.03915	2.00062

Bibliography

[1] AKAROA2. http://coscweb2.cosc.canterbury.ac.nz/research/RG/net_sim/simulation_group/akaroa

[2] PDNS. http://www.cc.gatech.edu/computing/compass/pads/projects.html

[3] Pawlikowski, K. "Simulation Studies of Telecommunication Networks and Their Credibility". *Proc. 13th European Simulation Multiconf.,* ESM'99 (Warsaw, Poland, 1999), SCS, 1999, 349-355.

[4] Ewing, G., Pawlikowski, K., McNickle D. "Akaroa2 : Exploiting Network Computing by Distributed Stochastic Simulation". *13th European Simulation Multiconf.,* (Warsaw, Poland, 1999), SCS, 1999, 175-181.

[5] Ewing, G., McNickle, D., Pawlikowski, K. "Multiple Replications in Parallel: Distributed Generation of Data for Speeding up Quantitative Stochastic Simulation". *Proc. 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics, Berlin,* August 1997, 379-402.

[6] Pawlikowski, K., V. Yau, D.McNickle. "Distributed Stochastic Discrete-Event Simulation in Parallel Time Streams". *Proc. Of the 1994 Winter Simulation Conf.,* WSC'94 (Orlando, USA, 1994), IEEE Press, 1994, 723-730.

[7] Pawlikowski, K., J. Jeong, R. Lee. "On Credibility of Simulation Studies of Telecommunication Networks". IEEE Communications Magazine, 40(1):132-139, Jan 2002.

[8] Pawlikowski, K. "Steady-State Simulation of Queueing Processes: A Survey of Problems and Solutions". *ACM Computing Surveys,* Vol.22, No.2, June 1990, 123-170.

[9] Bhavsar, V. C., Isaac, J. R. "Design and Analysis of Parallel Monte Carlo". *SIAM J. Sci. Stat. Computing,* 8 (1), 1987, 73-95.

[10] Rego, V. J., Sunderam, V. S. "Experiments in Concurrent Stochastic Simulation: the EcliPSe Paradigm". *Journal of Parallel and Distributed Computing,* 1992, 14: 66-84.

[11] Sunderam, V. S., Rego, V. J. "EcliPSe: A System for High Performance Concurrent Simulation". *Software-Practise and Experience,* 1991, 11: 1189-1219.

[12] Heidelberger, P. "Discrete Event Simulations and Parallel Processing: Statistical Properties". *SIAM J. Sci. Computing,* 8(6), 1988, 1114-1132.

[13] Heidelberger, P. "Statistical Analysis of Parallel Simulations". *Proc. Winter Simulation Conf.,* WSC'86, 1986, 290-295.

[14] Reynolds, P. "A Spectrum of Options for Parallel Simulation". *Proc. Winter Simulation Conf.,* WSC'88, 1988, 325-332.

[15] Righter, R., Walrand, J. "Distributed Simulation of Discrete Event Simulation". *Proc. The IEEE* vol. 77, No. 1, Jan 1989, 99-113.

[16] Fujimoto, R. "Parallel Discrete Event Simulation". *Communications of the ACM,* vol. 33, Oct 1990, 31-53.

[17] Fujimoto, R. Parallel and Distributed Simulation Systems, John Wiley Sons, 2000.

[18] Riley, G., Fujimoto, R., Ammar, M. H. 1999. "A Generic Framework for Parallelization of Network Simulations". *Seventh International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems.*

[19] Wu, H., Fujimoto, R., Riley, G. 2001. "Experiences Parallelizing A Commercial Network Simulator". *Proc. Of the 2001 Winter Simulation Conf.,* IEEE Press, 2001, 1353-1360.

[20] Riley, G., Fujimoto, R., Ammar, M. H. Xu, D., Wu, H. 2000. "Experiences Parallelizing Serial Network Simulations"

[21] The Network Simulator - ns-2. `http://www.isi.edu/nsnam/ns`

[22] Woesner, H., Gillich, E., Kopke, A. The NS-2/Akaroa-2 project. `http://www-tkn.ee.tu-berlin.de/research/ns-2_akaroa-2/ns.html`

[23] Woesner, H., Hoene, C. "Design and Verification of an IEEE 802.11e EDCF Simulation Model in ns-2.26." Technical Report, Technical University Berlin Telecommunication Networks Group, Nov 2003.

[24] Bruschi, S., Santana, M., Santana, R., Aiza, T. "An Automatic Distributed Simulation Environment". *Proc. Of the 2004 Winter Simulation Conf.,* 2004, 378-385.

[25] Conway, R. "Some Tactical Problems in Digital Simulation". *Management Sci.* 10, 1963, 47-61.

[26] Conway, R., Johnson, B., Maxwell, W. "Some Problems of Digital Systems Simulation". *Management Sci.* 6, 1959, 92-110.

[27] Stevens, W. "UNIX Network Programming". Volume 1, Prentice Hall PTR, 1998.

[28] Sauer, C., MacNair, E. "Simulation of Computer Communication Systems", Prentice Hall, Inc., 1983.

[29] A.M. Law, W.D. Kelton. "Simulation Modelling and Analysis. Third Edition", McGraw-Hill, New York, 2000.

[30] S. Bajaj, et al. "Improving Simulation for Network Research", USC Computer Science Department Technical Report 99-702, 1999.

[31] Rao, D. Thondugulam, N., Radhakrishnan, R., Wilsey, P. "Unsynchronised Parallel Discrete Event Simulation". *Prof. Of the 1998 Winter Simulation Conf.,* 1998, 1563-1570.

[32] Breslau et el. "Advances in Network Simulation", *IEEE Computer*, May,2000, 59-67.

[33] Meyer, R., Martin, J., Bagrodia, R. "Slow Memory: the Rising Cost of Optimism", *Proc. PADS 2000*, May 2000, 45-52.

[34] Ferenci, S., Penumalla, K., Fujimoto, R. "An Approach for Federating Parallel Simulators", *Proc. PADS 2000*, May 2000, 63-70.

[35] Gan, B.P. et al. "Load Balancing for Conservative Simulation on Shared Memory Multiprocessor Systems", *Proc. PADS 2000*, May 2000, 139-146.

[36] Floyd, S., Paxson, V. "Difficulties in Simulating the Internet", *IEEE/ACM Transactions on Networking*, Vol.9, No.4, Aug 2001, 392-403.

[37] Cowie, J., Nicol, D., Ogielski, A. "Modeling the Global Internet", *IEEE, Computing in Science Engineering*, Jan-Feb 1999, 42-50.

[38] Bagrodia, R. "Perils and Pitfalls of Parallel Discrete-Event Simulation", *Proc. of the 1996 Winter Simulation Conf.,* 1996, 136-143.

[39] Bryant, R.E. "Simulation of packet communications architecture computer systems", 1977,*MIT-LCS-TR-188.*

[40] Chandy, K.M., Misra, J. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs", *IEEE Transactions on Software Engineering* 1978, SE-5(5):440-452.

[41] Jefferson, D. "Virtual Time", *ACM Transactions on Programming Languages and Systems*, 1985, 7(3):404-425.

[42] Tropper, C. "Parallel and Distributed Discrete Event Simulation", Nova Science Publishers, Inc., 2002.

[43] Nicol, D.M., Reynolds, P.F., "Problem oriented protocol design.", *Proc. of the 1984 Winter Simulation Conf.*, pp. 471-474.

[44] Misra, J. "Distributed-discrete event simulation", *ACM Comput, Surv. 18*, 1, March 1986, 39-65.

[45] Lubachevsky, B.D. "Efficient distributed event-driven simulations of multiple-loop networks", *Commun. ACM 32*, Jan 1989, 111-123.

[46] Chandy, K.M., Sherman, R. "The conditional event approach to distributed simulation", *Proc. of the SCS Multiconference on Distributed Simulation 21*, 2 Mar 1989, pp 93-99.

[47] Gafni, A. "Rollback mechanisms for optimistic distributed simulation systems", *Proc. of the SCS Multiconference on Distributed Simulation 19*, 3, July 1988, pp. 61-67.

[48] West, D. "Opimizing Time Warp: Lazy rollback and lazy re-evaluation", M.S. thesis, Uni. of Calgary, Jan 1988.

[49] Sokol, L.M., Briscoe, D.P., Wieland, A.P. "MTW: a strategy for scheduling discrete simulation events for concurrent execution", *Proc. of the SCS Multiconference on Distributed Simulation 19*, 3, July 1988, pp.34-42.

[50] Madisetti, V., Walrand, J., Messerschmitt, D. "Wolf: A rollback algorithm for optimistic distributed simulation systems", *Proc. of the 1988 Winter Simulation Conf.* pp.296-305.

[51] Fujimoto, R.M. "Time Warp on a shared memory multiprocessor", *Trans. Soc. for Comput. Simul. 6*, 3, July 1989, 211-239.

[52] Chandy, K.M., Sherman, R. "Space, time, and simulation", *Proc. of the SCS Multiconference on Distributed Simulation 21*, 2, Mar 1989, pp. 53-57.

[53] Lubachevsky, B.D., Shwartz, A., Weiss, A. "Rollback sometimes works ... if filtered". *Proc. of the 1989 Winter Simulation Conf.* Dec 1989, pp.630-639.

[54] Reynolds, P.F. "A shared resource algorithm for distributed simulation" *Proc. of the 9th Annual Symposium on Computer Architecture, 10*, 3, Apr 1982, 259-266.

[55] Knop, F., Sunderam, V.S. "An introduction to fault tolerant parallel simulation with EcliPSe", *Proc. of the 1994 Winter Simulation Conf.* pp. 700-707.

[56] Preiss, B., Loucks, W. "Memory Management Techniques for Time Warp on a Distributed Memory Machine", *Proc. of 1995 Workshop on Parallel and Distributed Simulation*, June 1995, pp.30-39.

[57] Jefferson. "Virtual Time II: The cancelback protocol for storage management in distributed simulation", *Proc. of the 9th Ann, ACM Symp. on Principles of Distributed Computation*, Aug 1990, pp. 75-90.

[58] Jones, D.W. "An empirical comparison of priority-queue and event-set implementations", *Communications of the ACM, 29(4)*, Apr 1986, pp. 300-311.

[59] Ronngren, R., Ayani, R., Fujimoto, R., Das, S. "Efficient implementation of event sets in time warp", *Workshop on Parallel and Distributed Simulation (PADS), SCS Simulation Series*, vol 23, May 1993, pp. 101-108.

[60] Ronngren, R., Riboe, J., Ayani, R. "Fast implementation of the pending event set", '*Int'l Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, SCS Simulation Series*, Jan 1993.

[61] Brown, R. "Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem", *Comm. ACM Vol.31, No.10*, Oct 1988, pp. 1220-1227.

[62] Tanner, M. Practical Queueing Analysis, McGraw-Hill International (UK) Ltd. 1995.

[63] Chou, CC., Bruell, S., Jones D., Zhang W. "A Generalized Hold Model", *Prof. of the 1993 Winter Simulation Conf.*, 756-761.

[64] Vaucher, J.G. "On the distribution of event times for the notices in a simulatiion event list", *IN. FOR, 15,* 2(June), pp. 171-182.

[65] Nandy , Loucks. "On a parallel partitioning technique for use with conservative parallel simulation", *Proc. of the 7th Workshop on Parallel and Distributed Simulation*, 1993, pp. 43-51.

[66] Nicol, Reynolds. "A statistical approach to dynamic partitioning", *Proc. of the SCS Multi-Conference*, Jan 1985, 53-56.

[67] Reiher, Jefferson. "Dynamic load management in the Time Warp Operating System", *Trans. of the Society for Computer Simulation, 7(2)*, June 1990, pp. 91-120.

[68] Fujimoto, R., Nicol, D. "State of the art in parallel simulation", *Proc. of Winter Simulation Conference*, 1992, pp.246-254.

[69] Fujimoto et el., "Large-Scale Network Simulation: How Big? How Fast?", *Proc. of the 11th IEEE/ACM Inter. Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS'03)*, 2003.

[70] Fujimoto, R. "Distributed Simulation Systems", *Proc. of Winter Simulation Conference*, 2003, pp. 124-134.

[71] Schlagenhaft, R., Ruhwandl, M., Sporrer, C., Bauer, H. "Dynamic Load Balancing of a Multi-Cluster Simulator on a Network of Workstations", IEEE, 1995, pp. 175-180.

[72] Fujimoto et al. "Design and evaluation of the rollback chip: Special purpose hardware for Time Warp", *IEEE Transactions on Computers*, Jan 1992, 41(1):68-82.

[73] Reynolds, P.F. "A Shared Resource Algorithm for Distributed Simulation", *Proc. of the 9th Annual Int'l Comp Arch Conf*, Austin, Texas, Apr 1982, 259-266.

[74] GTNetS. http://www.ece.gatech.edu/research/labs/MANIACS/GTNetS.

[75] Dickens, P.M., Reynolds, P.F. "SRADS with local rollback", *Proc. of the SCS Multiconference on Distributed Simulation*, 22, 1, Jan 1990, 161-164.

[76] Fishwick P. "Web-based Simulation: Some Personal Observations", *Proc. of the 1996 Winter Simulation Conference*, 1996, 772-779.

[77] Veith T, et al. "World Wide Web-based Simulation", *Int. J. Engng Ed.* Vol.14, No.5, 1996, pp.316-324.

[78] Jefferson D., Reiher P. "Supercritical Speedup", *24th Annual Simulation Symposium*, 1991, 159-168.

[79] Nicol, D. "Scalability, Locality, Partitioning and Synchronization in PDES", Vol.28,1, July 1998 ACM SIGSIM Simulation Digest, 4-11.